ConceptBase.cc User Manual Version 7.3

Jeusfeld, M.A.; Quix, C.; Jarke, M.

Publication date:
2011

ConceptBase.cc User Manual
Version 7.3 [1]

Manfred A. Jeusfeld[+], Christoph Quix[*], Matthias Jarke[*]

[+] Tilburg University, Warandelaan 2, 5037AB Tilburg, Netherlands
[*] RWTH Aachen, Ahornstr. 55, 52056 Aachen, Germany

2011-02-25

Abstract. ConceptBase.cc (in short ConceptBase) is a multi-user deductive object manager intended for conceptual modeling, meta modeling, and coordination in design environments. The system implements O-Telos, a dialect of Telos integrating properties of deductive and object-oriented languages. It uniformly represents all information regardless of its abstraction level (data, class, meta class, meta meta class etc.) in a single data structure called P-facts. The powerful deductive query language is seamlessly integrated into the meta class hierarchy. Modeling is supported by meta classing, deduction and integry checking, active rule specification, functional definition of computation, a module concept, and a historical database allowing to query past states of the database. These principles are combined orthogonally, e.g. deductive rules can be contained to modules, formulated for meta classes, employed in active rules using functional definitions to compute properties, and be revised without overwriting the earlier definitions. The Java-based usage environment offers an extensible palette of graphical, tabular, and textual user interfaces. The communication between the user clients and the object base is organized in a client-server architecture using TCP/IP. Further clients can be connected by an API documented in the ConceptBase programmers manual.

Contributions to this manual were made by: Lutz Bauer, Rainer Gallersdörfer, Michael Gebhardt, Matthias Jarke, Manfred Jeusfeld, Thomas List, Hans Nissen, Christoph Quix, René Soiron, and Martin Staudt.

Major contributions to the source code of ConceptBase were made by: Lutz Bauer (module system), Rainer Gallersdörfer (object store), Manfred Jeusfeld (CB server, logic foundation, function component), Eva Krüger (integrity component), Thomas List (object store), Hans Nissen (module system), Christoph Quix (CB server, view component), Christoph Radig (object store), René Soiron (optimizer), Martin Staudt (CB server, query component), Kai von Thadden (query component), and Hua Wang (anwer formatting).

Additional contributions came from Masoud Asady, Markus Baumeister, Ulrich Bonn, Stefan Eherer, Michael Gebhardt, Dagmar Genenger, Michael Gocek, Rainer Hermanns, David Kensche, André Klemann, Rainer Langohr, Tobias Latzke, Xiang Li, Yong Li, Farshad Lashgari, Andreas Miethsam, Martin Pöschmann, Achim Schlosser, Tobias Schöneberg, Claudia Welter, Thomas Wenig, and others.

Contents

1 Introduction **5**
- 1.1 Background and history . . . 6
  - 1.1.1 Telos and O-Telos . . . 6
- 1.2 The architecture of ConceptBase.cc . . . 8
- 1.3 Hardware and software requirements . . . 8
- 1.4 Overview of this manual . . . 9
- 1.5 Differences to earlier versions . . . 10
- 1.6 License terms . . . 10

2 O-Telos by ConceptBase.cc **11**
- 2.1 Frame and network representation . . . 11
- 2.2 Rules and constraints . . . 17
  - 2.2.1 Basic predicates . . . 17
  - 2.2.2 Notes on attribution . . . 19
  - 2.2.3 Assigning attribute categories to explicit attributes . . . 21
  - 2.2.4 Reserved words . . . 22
  - 2.2.5 Comparison predicates . . . 22
  - 2.2.6 Typed variables . . . 22
  - 2.2.7 Semantic restrictions on formulas . . . 23
  - 2.2.8 Rule and constraint syntax . . . 25
  - 2.2.9 Meta formulas . . . 26
  - 2.2.10 Further object references . . . 28
  - 2.2.11 User-definable error messages for integrity constraints . . . 28
- 2.3 Query classes . . . 29
  - 2.3.1 Query definitions versus query calls . . . 33
  - 2.3.2 Query classes and deductive integrity checking . . . 34
  - 2.3.3 Nested query calls and shortcuts . . . 35
  - 2.3.4 Reified query calls . . . 36
- 2.4 View definitions . . . 37
- 2.5 Functions . . . 38
  - 2.5.1 Functions as special queries . . . 38
  - 2.5.2 Shortcuts for function calls and functional expressions . . . 39
  - 2.5.3 Example function calls and definitions . . . 40
  - 2.5.4 Programmed functions . . . 41
  - 2.5.5 Recursive function definitions . . . 42
- 2.6 Query evaluation strategies . . . 43
- 2.7 Datalog queries and rules . . . 45
  - 2.7.1 Extended query model . . . 45
  - 2.7.2 Datalog code . . . 45
  - 2.7.3 Examples . . . 46

3 Answer Formats for Queries **48**
3.1 Basic definitions . . . 48
3.2 Constructs in answer formats . . . 50
3.2.1 Simple expressions in patterns . . . 50
3.2.2 Pre-defined variables . . . 51
3.2.3 Iterations over expressions . . . 52
3.2.4 Special characters . . . 52
3.2.5 Function patterns . . . 52
3.2.6 Calling queries in answer formats . . . 53
3.2.7 Expressions in head and tail . . . 54
3.2.8 Conditional expressions . . . 54
3.2.9 Views and path expressions . . . 55
3.3 Parameterized answer formats . . . 55
3.4 File type of answer formats . . . 56

4 Active rules **57**
4.1 Definition of ECArules . . . 57
4.1.1 ECAassertion . . . 58
4.1.2 Events . . . 58
4.1.3 Conditions . . . 58
4.1.4 Actions . . . 59
4.1.5 Priorities . . . 60
4.1.6 Mode of an ECA rule . . . 60
4.1.7 Execution Semantics . . . 61
4.1.8 Activate and Deactivate ECA rules . . . 61
4.1.9 Depth . . . 62
4.1.10 User-definable Error Messages . . . 62
4.1.11 Constraints . . . 62
4.2 Examples . . . 62
4.2.1 Materialization of views by active rules . . . 62
4.2.2 Counter . . . 63
4.2.3 Timestamps . . . 64
4.2.4 Simulation of Petri Nets . . . 65
4.3 Limitations in the current implementation . . . 65

5 The Module System **67**
5.1 Definition of modules . . . 67
5.2 Switching between module contexts . . . 68
5.3 Using nested modules . . . 69
5.4 Exporting and importing objects . . . 69
5.5 Setting user home modules . . . 70
5.6 Limiting access to modules . . . 71
5.7 Listing the module content . . . 74
5.8 Saving and loading module sources . . . 75
5.9 Server-side materialization of query results . . . 76

6 The ConceptBase.cc Server **77**
6.1 CBserver parameters . . . 77
6.2 The cache subsystem . . . 80
6.3 Database persistency . . . 81
6.4 The UNTELL operation . . . 81
6.5 Memory consumption and performance . . . 83

7 The CBshell utility **84**
7.1 Syntax . . . 84
7.2 Options . . . 84
7.3 Commands . . . 84

8 The ConceptBase Usage Environment **86**
8.1 CBiva . . . 86
8.1.1 The menu bar . . . 87
8.1.2 The tool bar . . . 89
8.1.3 The status bar . . . 89
8.1.4 Telos editor . . . 89
8.1.5 History window . . . 90
8.1.6 Display instances . . . 90
8.1.7 Frame browser . . . 90
8.1.8 Display queries . . . 90
8.1.9 Query editor . . . 92
8.1.10 Tree browser . . . 92
8.2 ConceptBase graph editor . . . 93
8.2.1 Overview . . . 93
8.2.2 Starting the graph editor . . . 95
8.2.3 Menu bar . . . 95
8.2.4 Tool bar . . . 97
8.2.5 Popup menu . . . 98
8.2.6 Editing of Telos objects . . . 99
8.2.7 Caching of query results within the graph editor . . . 101
8.3 An example session with ConceptBase . . . 102
8.3.1 Starting ConceptBase . . . 102
8.3.2 Connecting CBiva to the ConceptBase server . . . 102
8.3.3 Loading objects from external files . . . 103
8.3.4 Displaying objects . . . 104
8.3.5 Browsing objects . . . 104
8.3.6 Editing Telos objects . . . 106
8.3.7 Using the query facility . . . 108
8.4 Usage as applet . . . 109
8.5 Configuration file . . . 112

A Syntax Specifications **116**
A.1 Syntax specifications for Telos frames . . . 116
A.2 Syntax of the rule and constraint language . . . 117
A.3 Syntax of active rules . . . 119
A.4 Terminal symbols . . . 120
A.5 Syntax specifications for SML fragments . . . 120

B O-Telos Axioms **123**

C Graphical Types **125**
C.1 The graphical type model . . . 125
C.2 The standard graphical types . . . 126
C.2.1 The extended graphical type model . . . 126
C.2.2 Default graphical types . . . 127
C.3 Customizing the graphical types . . . 127
C.3.1 Properties of CBIndividual and CBLink . . . 127
C.3.2 Providing a new implementation for a graphical type . . . 128
C.3.3 Shapes . . . 130

D Examples **131**
D.1 Example model: The Employee model . . . 131
D.2 A Telos modeling example - ER diagrams . . . 131
D.2.1 The basic model . . . 131
D.2.2 The use of meta formulas . . . 133
D.2.3 Limitations and final remarks . . . 138

E Predefined Query Classes **140**
E.1 Query classes and generic query classes . . . 140
E.1.1 Instances and classes . . . 140
E.1.2 Specializations and generalizations . . . 140
E.1.3 Attributes . . . 141
E.1.4 Links between objects . . . 141
E.1.5 Other queries . . . 141
E.2 Functions . . . 141
E.2.1 Computation and counting . . . 142
E.2.2 String manipulation . . . 142
E.3 Builtin query classes . . . 143

F CBserver Plug-Ins **144**
F.1 Defining the plug-in . . . 144
F.2 Calling the plug-in . . . 144
F.3 Programming interface for the plug-ins . . . 145

Chapter 1

Introduction

ConceptBase is a deductive object base management system for meta databases. Its data model is a conceptual modeling language making it particularily well-suited for design and modeling applications. Its underlying data model allows to uniformly represent data, classes, meta classes, meta meta classes etc. yielding a powerful meta-CASE environment. The system has been used in projects ranging from development support for data-intensive applications [JMSV90, JJQV99, JQC*00], requirements engineering [RaDh92, Eber97, NJJ*96], electronic commerce [QSJ02], and version&configuration management [RJG*91] to co-authoring of technical documents [HJEK90].

The key features distinguishing ConceptBase from other extended DBMS and meta-modeling systems are:

- Unlimited meta class hierarchy, allowing to represent information at any abstraction level (data, class, meta class, meta meta class)
- Uniform data structure (called *P-fact*) for concepts, their attributes, their class memberships, and their super- and sub-concepts; all four types of information are full-fledged objects
- Clean formal integration of deductive and object-oriented abstraction by Datalog logical theories
- Complex computations can be user-defined by recursive function definitions, e.g. the length of the shortest path between two nodes
- Queries are defined as classes with user-defined membership constraints; queries can range over any type of object at any abstraction level
- Active rules can be used to define the system's reaction to events; active rules can change the state of the database and can trigger each other
- Client-server architecture with wide-area Internet access

ConceptBase implements the version O-Telos (= Object-Telos) of the knowledge representation language Telos [MBJK90]. O-Telos integrates a thoroughly axiomatized structurally object-oriented kernel with a predicative assertion language in the style of deductive databases. A complete formal definition can be found in [JGJ*95, Jeus92]. O-Telos is viewed as a hybrid knowledge representation language integrating ideas from frame-like concept description languages and deductive databases. This hybrid character suggests a hypertext-style combination of (graphical) semantic network views and (textual) frame representations at the user interface. Both O-Telos views are based on the common logical representation so that they can be treated in a completely symmetrical way, both for purposes of querying/browsing and of editing/manipulation.

This manual is integrated with the ConceptBase Forum. The ConceptBase Forum is an Internet-based workspace where ConceptBase developers and users share knowledge. It contains numerous examples on how to solve certain modeling problems. It is highly recommended to join the workspace. More details are available at `http://conceptbase.sourceforge.net/CB-Forum.html`.

ConceptBase is mainly used for metamodeling and for engineering customized modeling languages. The textbook [JJM*09]

```
Jeusfeld, M.A., M. Jarke, and J. Mylopoulos:
Metamodeling for Method Engineering.
Cambridge, MA, 2009. The MIT Press, ISBN-10: 0-262-10108-4.
```

introduces into the topic and provides six in-depth case studies ranging from requirements engineering to chemical device modeling. The book and this user manual are complementary to each other.

1.1 Background and history

Proposals for next-generation databases [BMS84] stem from three traditions:

- *Databases:* This stream is best typified by extended relational and object-oriented databases. The main extension is the addition of complex domains, shared sub-objects, and procedures to the database. Examples include Postgres, Damokles, Cactis, Iris, DASDBS, and many others.
- *Programming languages:* The main goal is to provide persistence and sharing to one or more programming languages, ideally orthogonal to the type systems of these languages. Examples include the O_2 system for imperative programming languages, and typed versions of Prolog such as LOGIN or PROTOS-L.
- *Knowledge representation:* Some of these systems aim at providing database service to AI applications, others come from the tradition of deductive databases or semantic data models and aim at formal as well as practical support for general database systems and applications. Examples include the deductive databases EKS, LDL, CORAL, NAIL!, and the concept logic systems CLASSIC, BACK, and LOOM.

A sample of the former two groups of systems is described in [TKDE90], whereas several prototypes of the latter are reported in [SIGA91]. Although a certain confluence can be observed, these systems do not only differ in their background theories but also in their intended application domains.

The first group wants to support non-standard applications such as the handling of complex engineering objects. The second group aims at an easier programming environment for standard applications: additionally, they emphasize the goal of making applications written in different languages interoperable.

The third group pursues, as one of its goals, the support for AI applications such as natural language understanding or expert systems. However, this may remain a fairly limited portion of the software market. The reason why we became interested in this group of languages is therefore a slightly different one: the important role we expect them to play in *meta data management*. Meta data applications range from the uniform access to heterogeneous data sources, to the coordination of design processes, to the integration of heterogeneous information services in networked production chains, whole enterprises, or even transnational networks. In these applications which are crucial for the huge integration tasks to be tackled in the 1990s, the possibility not only to *execute* systems but also to *reason* formally about their structure and capabilities, can be considered a competitive advantage over the other approaches. *ConceptBase* has provided some validity to this claim by extensive usage experiments in several of the above areas.

1.1.1 Telos and O-Telos

The language underlying ConceptBase, *Telos*, has been one of the earliest attempts to integrate deductive and object-oriented data models [Stan86, MBJK90]. The O-Telos [Jeus92] dialect supported in ConceptBase takes a conservative integration approach, with the main design goals of semantic simplicity, symmetry of deductive and object views, flexibility and extensibility. This emphasis, technically supported by a careful mapping of Telos to Datalog with negation, has paid off both in user acceptance and ease of implementation. In essence, ConceptBase is based on deductive database technology woth object-oriented abstraction principles like object identity, class membership, specialization and attribution being coded as pre-defined deductive rules and integrity constraints.

Development of ConceptBase started in late 1987 in the context of ESPRIT project DAIDA [Jark93] and was continued within ESPRIT Basic Research Actions Compulog 1 and 2 (1989 – 1995), the ESPRIT LTR project DWQ (Foundations of Data Warehouse Quality, 1996-1999), and the ESPRIT project MEMO (Mediating and Monitoring Electronic Commerce, 1999-2001). Versions have been distributed for research experiments since early 1988. ConceptBase has been installed at more than eight hundred sites worldwide and is seriously used by about a dozen research projects in Europe, Asia, and the Americas.

The direct predecessor of O-Telos is the knowledge representation language Telos (specified by John Mylopoulos, Alex Borgida, Manolis Koubarakis, and others). Telos was designed to represent information about information systems, esp. requirements. Telos was based on CML (Conceptual Modeling Language) developed in the mid/late 1980-ties. A variant of CML was created under the label SML (System Modeling Language) and implemented by John Gallagher and Levi Solomon at SCS Hamburg. CML itself was based on RML (Requirements Modeling Language) developed at the University of Toronto by Sol Greenspan and others. Neither RML nor CML were implemented in 1987. They were regarded as theoretic knowledge representation languages with 'possible world semantics'. SML was implemented as a subset of CML using Prolog's SLDNF semantics.

In 1987, we decided to start an implementation of Telos and quickly realized that the original semantics was too complex for an efficient implementation. The temporal component of Telos included both valid time (when an information is true in the domain) and transaction time (when the information is regarded to part of the knowledge base). The temporal reasoner for the valid time was based on Allen's interval calculus and was co-NP-hard. We implemented both temporal dimensions in ConceptBase V3.0 only to see that there were undesired effects with the query evaluator and the uniform representation of information into objects. Specifically, the specialization of a class into a subclass could have a valid time which could be incomparable to the valid time of an instance of the subclass. Any change in the network of valid time intervals could change the set of instances of a class. Because of that, we dropped the valid time as a built-in feature of objects but we kept the transaction time. A few other features like the declarative TELL, UNTELL and RETELL operations as proposed by Manolis Koubarakis in his master thesis on Telos were only implemented in a rather limited way - essentially forbidding direct updates to derived facts. On the other hand, O-Telos extends the universal object representation to any piece of explicit information and reduces the number of essential builtin objects to just five. So, some of the roots of Telos in artificial intelligence were abandoned in favor of a clear semantics and of better capabilities for meta modelling.

Since we wanted to be able to manage large knowledge bases (millions of concepts rather than a few hundred), we decided to select a semantics that allowed efficient query evaluation. Telos included already features for deductive rules and integrity constraints. Thus, the natural choice was DATALOG with perfect model semantics. The deductive rule evaluator and the integrity checker were ready in 1990. A query language ("query classes") followed shortly later. O-Telos exhibits an extreme usage of DATALOG:

- There is only one base relation P(o,x,l,y) called P-facts used for all objects, classes, meta classes, attributes, class membership relationships, and specialization relationships as well as for deductive rules, integrity constraints and queries.
- The semantics of class membership, specialization, and attribution is encoded by around 30 axioms, which are either deductive rules or integrity constraints.
- Deductive rules ranging over more than one classification level (instances, classes, meta classes, etc.) are partially evaluated to a collection of rules (or constraints) ranging over exactly one classification level.

O-Telos should be regarded as the data modeling language of a meta database. It is capable to represent semantic features of other (data) modeling languages like entity-relationship diagrams and data flow diagrams. Once modeled as meta classes in O-Telos, one simply has to tell the meta classes to ConceptBase to get an evironment where one can manipulate models in these modeling languages. Since all abstraction levels are supported, the models themselves can be represented in O-Telos (and thus be managed by ConceptBase).

ConceptBase's implementation of O-Telos provides a couple of features beyond the core O-Telos. First of all, there is a dedicated query language CBL, which provides a class-based interpretation for queries.

Secondly, modules have been introduced to structure the search space. Essentially, the module identifier is added to the object identifier. Thirdly, ConceptBase supports a limited version of active rules to react to internal and external events. Finally, ConceptBase supports recursively defined functions and arithmetic expressions.

1.2 The architecture of ConceptBase.cc

ConceptBase.cc 7.3 follows a client-server architecture. Clients and servers run as independent processes which interact via inter-process communication (IPC) channels (Fig. 1-1). Although this communication channel was initially meant for use in local area networks, it has been used successfully for nationwide and even transatlantic collaboration of clients on a common server.

The ConceptBase.cc server (CBserver) offers programming interfaces that allow to build clients and to exchange messages in particular for updating and querying object bases using the Telos syntax. We provide support for C, Java, TCL, and Prolog. Detailed descriptions of the interfaces and the corresponding libraries that are delivered with ConceptBase.cc can be found in the **ConceptBase Programmers Manual**.

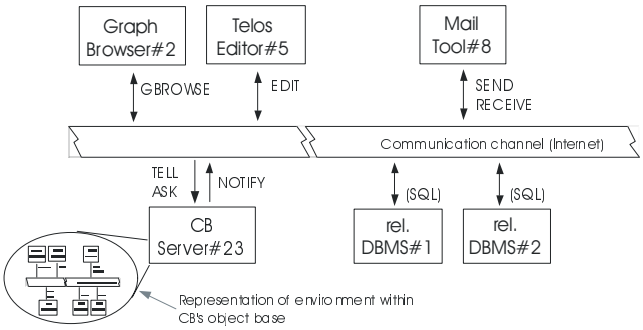

Figure 1.1: The client-server architecture of ConceptBase.cc

ConceptBase.cc comes with a standard graphical usage environment (*CBjavaInterface*) implemented in Java which supports editing, ad-hoc querying and browsing of O-Telos object bases. The user interfaces of ConceptBase can serve for the user as an example client for programming own application specific client tools. Although ConceptBase.cc provides multi-user support and an arbitrary number of clients may be connected to the same server process, ConceptBase.cc does not yet support concurrency control beyond a forced serialization of messages.

1.3 Hardware and software requirements

ConceptBase.cc operates[1] on

- SUN SPARC CPUs under Solaris 8 or higher,
- i386 CPUs under Solaris 8 or higher,
- i386 CPUs under Linux Kernel 2.4 or higher,
- i64 (AMD64) CPUs under Linux Kernel 2.4 or higher,
- i386 CPUs under Microsoft Windows 2000/XP/Vista/7.

The platform Mac OS-X on PowerPC CPUs is no longer supported by ConceptBase.cc. The last available built for this platform is ConceptBase 7.1, compiled January 18, 2008. This built is available from the ConceptBase Forum.

[1] All trademarks are property of their respective owners.

Implementation languages are Prolog[2] (in particular for logic-based transformation and compilation tasks) and C/C++ (in particular for persistent object storage and retrieval). The ConceptBase.cc usage environment (CBiva) executes on any platform with a compatible Java Virtual Machine (Java 1.4, Java 5, Java 6 Update 11 or earlier, OpenJDK Java 6 Runtime). We suggest to use the Java 5 Runtime Environment, which is available at Sun's Java Archive [http://java.sun.com/products/archive/]. If you downloaded the Windows variant of ConceptBase.cc, you do not need to install a compatible Java Runtime Environment since it is already included in the binaries.

The ConceptBase.cc server (CBserver) is dynamically linked with a couple of shared libraries. Under Linux 2.6, you should make sure that the following libraries are installed. Most of them will be already present on a standard Linux installation.

```
linux-gate.so.1,libstdc++.so.6,libgmp.so.3,librt.so.1
libreadline.so.5,libncursesw.so.5,libm.so.6
libdl.so.2,ibgcc_s.so.1,libpthread.so.0,libc.so.6
/lib/ld-linux.so.2,libncurses.so.5
```

The requirements for other Unix-style platforms are similar. Following is the list of shared libraries expected under Solaris 9:

```
libnsl.so.1,libsocket.so.1,libstdc++.so.5
librt.so.1,libdl.so.1,libcurses.so.1,libm.so.1
libc.so.1,libmp.so.2,libgcc_s.so.1
libaio.so.1,libmd5.so.1
/usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1
/usr/platform/SUNW,Ultra-5_10/lib/libmd5_psr.so.1
```

The installation of ConceptBase.cc requires about 50 MB of free hard disk space. The main memory requirements depend on the size of the object base loaded to the ConceptBase.cc server. The initial main memory footprint is just about 8 MB. We recommend about 20 MB of free main memory for small applications and 200 MB and higher for large applications of ConceptBase.cc. The server can handle relatively large databases consisting of a few million objects. Response times depend on the size of the database and even more on the structure of the query.

Since clients connect to a CBserver via Internet, the server requires the TCP/IP protocol to be available on both the client and the server machine (can be the same computer for single-user scenarios). Note that a firewall installed on the path between the client and the server machine might block remote access to a CBserver. The default port number used for the communication between server and client is 4001. It can be set to another port number by a command line parameter.

The CBserver is by default multi-user capable, i.e. multiple clients can connect to the same CBserver. This feature is by default disabled when you start the CBserver from within the user interface. See section 6 for more details.

The standard ConceptBase.cc client is CBiva (see section 8). The distribution also contains a client CBshell that can be used to interaction with a ConceptBase server using a command/shell window. The CBshell client can also be used to run non-interactive scripts, e.g. for loading a sequence of files with concept definitions into the CBserver. Access to a CBserver via a web interface is also supported. Required software is made available via the ConceptBase Forum.

1.4 Overview of this manual

This manual provides detailed information about using ConceptBase.cc. Information about the installation procedure can be found in the Installation Guide in directory doc/TechInfo. New users are advised to follow the installation guide for getting the system started and then to work through the ConceptBase.cc Tutorial. More information about the knowledge representation mechanisms, the applications, and the

[2]ConceptBase.cc now relies on SWI-Prolog [http://www.swi-prolog.org/]. Formerly, ProLog by BIM had been used.

implementation concepts can be found in the references. Chapter 2 describes the ConceptBase.cc version of the Telos language and gives some examples for its usage. Chapter 6 discusses the parameters that can be set when starting the CBserver. Finally, section 8 describes the ConceptBase.cc Usage Enviroment.

Appendices contain a formal definition of the Telos syntax and internal data structures (A). Appendix C summarizes the mechanism for assigning graphical types to objects and adapting the graphical browsing tool for specific application needs. Appendix D contains the full Telos notation of an example model (D.1) and a case study on the modeling of entity-relationship diagrams with Telos (D.2). Plenty of further examples for particular application domains and add-ons for meta modeling can be retrieved from the ConceptBase Forum.

1.5 Differences to earlier versions

ConceptBase.cc 7.3 should be largely compatible to its direct predecessor. We mainly improved the stability and removed some restrictions of the metaformula compiler.

The release notes to ConceptBase.cc 7.3 lists all major changes and issues. You find the release notes in the subdirectory `doc/TechInfo` of your ConceptBase.cc installation directory or via the web site http://conceptbase.cc.

The system still has about the same memory footprint as it used to be 10 years ago. You can easily install the complete system for all supported platforms on a 32 MB memory stick.

1.6 License terms

ConceptBase.cc is distributed under a FreeBSD-style copyright license since June 2009. Both binary and source code are available via http://sourceforge.net/projects/conceptbase/files.

The FreeBSD-style copyright license of ConceptBase.cc reads like follows:

```
The ConceptBase.cc Copyright

Copyright 1987-2011 The ConceptBase Team. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted
provided that the following conditions are met:

   1. Redistributions of source code must retain the above copyright notice, this list of
      conditions and the following disclaimer.
   2. Redistributions in binary form must reproduce the above copyright notice, this list of
      conditions and the following disclaimer in the documentation and/or other materials
      provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE CONCEPTBASE TEAM ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE CONCEPTBASE TEAM OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors
and should not be interpreted as representing official policies, either expressed or implied,
of the ConceptBase Team.
```

This license makes ConceptBase.cc free software as promoted by the Free Software Foundation. The license is compatible to the GNU Public License (GPL) and many other free license models. You are welcome to contribute the ConceptBase.cc project! Join the ConceptBase Forum at http://conceptbase.sourceforge.net/CB-Forum.html to do so.

Chapter 2

O-Telos by ConceptBase.cc

ConceptBase.cc is an implementation of the O-Telos data model. O-Telos is derived from the knowledge representation language Telos as designed by Borgida, Mylopoulos and others [MBJK90]. While Telos was geared more to its roots in artificial intelligence, O-Telos is more geared to database theory, in particular to deductive databases. Nevertheless, O-Telos is to a large degree compatible to the original Telos specification. In some respects, it generalizes Telos, for example by removing the requirement to classify objects into the levels for tokens, simple classes, and meta classes. In O-Telos, we have just five predefined objects (see appendix on the axioms of O-Telos).

Telos (and O-Telos) as well also have strong links to the semantic web, in particular to the triple predicates used for defining RDF(S) statements. The main difference is that O-Telos is based on quadruples where the additional components identifies the statement. While RDF(S) has to use special link types to reify triple statements, i.e. to make statements about statements, O-Telos statements are simply referred to by their identifier.

Telos' structurally object-oriented framework generalizes earlier data models and knowledge representation formalisms, such as entity-relationship diagrams or semantic networks, and integrates them with predicative assertions, temporal information, and in particular meta modeling. This combination of features seems to be particularly useful in software information applications such as requirements modeling and software process control. A formal description of O-Telos can be found in [MBJK90, Jeus92]. The following example is used throughout this section to illustrate the language:

> *A company has* `employees`*, some of them being* `managers`*. Employees have a* `name` *and a* `salary` *which may change from time to time. They are assigned to* `departments` *which are* `headed` *by managers. The* `boss` *of an employee can be derived from his department and the manager of that department. No employee is allowed to earn more money than his boss.*

This section is organized as follows: first, the frame and network representation of the O-Telos language is explained. Then, the predicative sublanguage for deductive rules and integrity constraints are presented. Subsection 2.3 presents a declarative query language which introduces queries as classes with optional predicative membership specification.

2.1 Frame and network representation

As a hybrid language O-Telos supports three different representation formats: a logical, a graphical (semantic network) and a frame representation. The two latter formats are based on the logical one. As explained in the next subsections the logical representation also forms the base for integrating a predicative assertion language for deductive rules, queries, and integrity constraints into the frame representation. We start with the so-called P-fact representation of a O-Telos object base.

An O-Telos object base is a finite set of interrelated propositions (=P-facts=objects):

$$OB = \{\mathtt{P}(\mathtt{oid}, \mathtt{x}, \mathtt{l}, \mathtt{y}, \mathtt{tt}) | \mathtt{oid}, \mathtt{x}, \mathtt{y}, \mathtt{tt} \in ID, \mathtt{l} \in LABEL\}$$

where oid has key property within the knowledge base, ID is a non-empty set of identifiers with a non-empty subset $LABEL$ of names. The components oid, x, l, y, tt are called identifier, source, label (or name), destination and belief time of the proposition[1]. We read them as follows:

> The object x has a relationship called l to the object y. This relationship is believed by the system for the time tt.

As shown below there is a natural interpretation of a set of propositions as a directed graph (semantic network). We distinguish four patterns of propositions and give them the following names:

Individuals
P(oid,oid,l,oid,tt)
("oid is an object with name l believed tt")

InstanceOf relationships (instantiations)
P(oid,x,*instanceof,y,tt)
("x is an instance of class y believed tt")

IsA relationships (specializations)
P(oid,x,*isa,y,tt) and
("x is a specialization of y believed tt")

Attributes (all other propositions)
P(oid,x,m,y,tt) and
("x has an attribute with label m with value y believed tt")

O-Telos imposes some structural axioms on knowledge bases, e.g. referential integrity, correct instantiation and inheritance ([Jeus92]). The complete list of axioms is contained in appendix B. The axioms are linked to predefined objects that are part of each O-Telos object base:

- Individual contains all individuals as instances. Intuitively, individuals are objects that are displayed as nodes in a graph.
- InstanceOf contains all explicit instantiation objects as instances. In the graphical representation, an instantiation object is a link between some object x and its class c.
- IsA contains all explicit specialization objects as instances. A specialization object is graphically displayed as a link between a subclass c and its superclass d.
- Attribute contains all explicit attribute objects as instances. An attribute object can be displayed by a link between an object x and the object y. The label of the attribute object is the name of the attribute link starting from x. The object y is also called the value of the attribute.
- Proposition contains all propositions as instances, i.e. the union of the instances of Individual, InstanceOf, IsA, and Attribute. The term proposition is used as a synonym to object in O-Telos. Hence, all explicit information in O-Telos is represented as proposition.

ConceptBase allows to derive instantiation of an object to a class and attributes relations between objects. This derived information has no object property, i.e. it is not identified and it is not represented as a proposition.

Additional to the above predefined classes[2], there are the builtin classes Class, Integer, Real and String. Class contains all so-called classes (including itself) as instances: the only special property of Class is the definition of two attribute categories rule and constraint. Hence, instances of

[1] We will see in section 2.2 that the predicative language operates on a snapshot of the object base, i.e. on those propositions that are believed at a specified reference time called rollback time. This time is an interval. The of the interval is the time when the object has been told/created. The end of the interval is either the time when the object was untold/deleted, or it is a special symbol Now indicating that the object is currently believed, i.e. it is not deleted.

[2] Strictly speaking, we should better use the term predefined object or predefined proposition.

classes can have deductive rules and integrity constraints. Integer and real numbers are written in the usual way, strings are character sequences, e.g. `"this is a string"`. These three classes are supported by comparison predicates like `(x < y)` discussed in section 2.2, and by functions like PLUS, MINUS discussed in section 2.5.

As legacy support, ConceptBase provides the pre-defined classes `Token`, `SimpleClass`, `MetaClass`, and `MetametaClass` to structure the object base into objects that have no instances (tokens), objects that have only tokens as instances (simple classes), objects that have only simple classes as instances (meta classes), and finally objects that have only meta classes as instances (meta meta classes). These classes are provided only for compatibility with older Telos specifications. In fact, an absolute hierarchy from tokens to simple classes to meta classes etc. is not an essential ingredient of O-Telos and in many situations too restrictive.

Instead, meta class levels are implicitely expressed via instantiation. If an object `x` is an instance of object `c` and object `c` is an instance of object `mc`, then `mc` is also called a *meta class* of `x`, and `c` a *class* of `x`. Being a class or a meta class is **relative** to the object `x` that we consider. For example, `mc` is the class of `c`. This implicit definition of the meta class concept is far more flexible than a fixed structure:

1. There is virtually no limit in the meta class hierarchy: there can be meta classes, meta meta classes, meta meta meta classes etc.

2. A class can have object from different meta class levels as instances. This is in particular important for extending the capabilities of the O-Telos language. An example of a class that has objects from different levels as instances is `Proposition`: it has *all* objects as instances.

3. A user does not need to decide to which meta class level an object belongs.

Strict conformance to the membership to meta class levels can still be enforced by user-definable integrity constraints.

As a user, you don't work directly with propositions but with textual (frame) and graphical (semantic networks) views on them. Both are not based on the oid's of objects but on their label components. To guarantee a unique mapping we need the following naming axiom:

Naming axiom (see also axioms 2,3,4 in appendix B)

1. The label ("name") of an individual object must be unique, i.e. if two objects have the same label than they are the same.
2. The label of an attribute must be unique within all attributes with a common source object, i.e. no two explicit attributes of the same object can have the same label. However, two different objects can well have attributes sharing the same label.
3. The source and destination of an instantiation object are unique, i.e. between two objects `x` and `y` may be at most one explicit instantiation link.
4. The source and destination of a specialization object are unique.

The **frame syntax** of O-Telos groups the labels of propositions with common source `o` around the label of `o`. The exact syntax is given in appendix A. In this section we introduce it by modeling the employee example:

```
Employee in Class with
   attribute
      name: String;
      salary: Integer;
      dept: Department;
      boss: Manager
end

Manager in Class isA Employee end
```

```
Department in Class with
   attribute
      head: Manager
end
```

The label of the "common source" in the first frame is `Employee`. It is declared as instance of the class `Class` and has four attributes. The class `Manager` is a subclass of `Employee`.

Oid's (preceded by '#' in our examples) are generated by the system. This leads to the following set of propositions corresponding to the frames above. The belief time inserted by the system is denoted by omission marks.

```
P(#E,#E,Employee,#E,...)
P(#1,#E,*instanceof,#Class,...)
P(#3,#E,name,#String,...)
P(#4,#E,salary,#Integer,...)
P(#5,#E,dept,#D,...)
P(#6,#E,boss,#M,...)
P(#M,#M,Manager,#M,...)
P(#7,#M,*instanceof,#Class,...)
P(#8,#M,*isa,#E,...)
P(#D,#D,Department,#D,...)
P(#9,#D,*instanceof,#Class,...)
P(#10,#D,head,#M,...)
```

Instantiation to the pre-defined class `Individual` is implicitly given by the structure of the three individual propositions named `Employee`, `Manager`, and `Department`. Analogously, the attributes `#3`, `#4`, `#5`, `#6` and `#10` are automatically regarded as instances of the class `Attribute`. The instances of `Attribute` are also called *attribution objects* or *explicit attributes*. Propositions `#1`, `#2`, `#7` and `#9` are instances of the class `InstanceOf` (holding explicit instantiation objects), and `#8` is an instance of the class `IsA` (explicit specialization objects). Note that all relationships are declared by using the identifiers (not the names) of objects. Thus, `#Class`, denotes the identifier of the object `Class` etc.

The identifiers are maintained internally by ConceptBase's object store. Externally, the user refers to objects by their name. A standard way to describe objects together with their classes, subclasses, and attributes is the frame syntax. Frames are uniformly based on object names.

The next frames establish two departments labelled `PR` and `RD` and state that the individual object `mary` is an instance of the class `Manager`. `Mary` has four attributes labelled `hername`, `earns`, `advises` and `currentdept` which are instances of the respective attribute classes of `Employee` with labels `name`, `salary` and `dept`.

```
mary in Manager with
  name
    hername: "Mary Smith"
  salary
    earns: 15000
  dept
    advises:PR;
    currentdept:RD
end

PR in Department end

RD in Department end
```

The corresponding propositions for the frame describing mary are:

```
P(#mary,#mary,mary,#mary,...)
P(#E1,#mary,*instanceof,#M,...)
P(#E3,#mary,hername,"Mary Smith",...)
P(#E4,#E3,*instanceof,#3,...)
P(#E5,#mary,earns,15000,...)
P(#E6,#E5,*instanceof,#4,...)
P(#E7,#mary,advises,#PR,...)
P(#E8,#E7,*instanceof,#5,...)
P(#E10,#mary,currentdept,#RD,...)
P(#E11,#E10,*instanceof,#5,...)
```

The attribute categories `name, salary` and `dept` must be defined in one of the classes of `mary`. In this case mary is also instance of `Employee` due to the following axiom which defines the inheritance of class membership in O-Telos, and hence can instantiate these attributes:

Specialization axiom (axiom 13 in appendix B)

> The destination ("superclass") of a specialization inherits all instances of its source ("subclass").

An example is the specialization #8: all instances of `Manager` (including `mary` are also instances of `Employee`. O-Telos enforces **typing** of the attribute values by the following general axiom:

Instantiation axiom (axiom 14 in appendix B)

> If p is a proposition that is instance of a proposition P then the source of p must be an instance of the source of P, and the destination of p must be an instance of the destination of P.

For example, "`Mary Smith`" must be an instance of `String`. The individual `mary` also shows another feature: attribute classes specified at the class level do not need to be instantiated at the instance level. This is the case for the `boss` attribute of `Employee`. On the other hand, they may be instantiated more than once as e.g. `dept`.

In some cases for attribute categories occuring in a frame the corresponding objects which are instantiated by the concrete attributes, can not uniquely be determined[3]. This multiple generalization/instantiation problem is solved[4] by the following condition which must hold for O-Telos object bases:

Multiple generalization/instantiation axiom (axiom 17 in appendix B)

> If p1 and p2 are attributes of two classes c1 and c2 which have the same label component l, and i is a common instance of c1 and c2 which has an attribute with category l, then there must exist a common specialization c3 of c1 and c2 with an l labelled attribute p3 which specializes p1 and p2, and i is instance of c3. Particularly if c1 is specialization of c2 and p1 is specialization of p2, c1 and p2 already fulfill the conditions for c3 and p3.

O-Telos treats all three kinds of relationships (`attribute`, `isa`, `in`) as objects. Thus each attribute, instantiation or generalization link of `Employee` may have its own attributes and instances. For example, each of the four `Employee` attributes is an instance[5] of an attribute class denoted by the label `attribute` but can also have instances of its own. The attribute with label `earns` of `mary` is an instance of attribute `salary` of class `Employee`. Syntactically, attribute objects are denoted by appending the attribute label with an exclamation mark to the name of some individual. The relationship between `salary` and `earns` could be expressed as

[3] Subsection 2.2 contains an example for this problem in the context of linking logical formulas to O-Telos objects.

[4] For specialization relationships between two objects we need an axiom similar to the instantiation axiom which requires specialization relationships between their sources and destination components. [Jeus92] contains the complete axiomatization.

[5] These instantiations were left out in the set of propositions for the employee example above.

```
mary!earns in Employee!salary
end
```

Instantiation links are denoted by the operator ”−>” and specialization links by ”=>”. They should always be enclosed in parentheses:

```
(mary->Manager)
end

(Manager=>Employee)
end
```

The operators can be combined to form complex expressions. The next example shows how to reference the instantiantion link between the attribute `mary!earns` and its attribute class `Employee!salary`. The second frame shows that arbitrarily complex expressions are possible. The parentheses have to be used to make the operator expressions unique. The attribution operator ”!” has a stronger binding than the instantiation and specialization operators. According to our own experience, complex expressions for denoting objects are rare in modeling. It is good to know that any object in O-Telos can be uniquely referenced in the frame syntax.

```
(mary!earns->Employee!salary) with
  comment
    com1: "This is a comment to an instantiation between attributes"
end

(mary!earns->Employee!salary)!com1 with
  comment
   com2: "This is a comment to the the previous comment attribute"
end
```

Figure 2.1 shows as the second view on propositions the graphical *semantic network representation* of mary and her relationships to the other example objects.

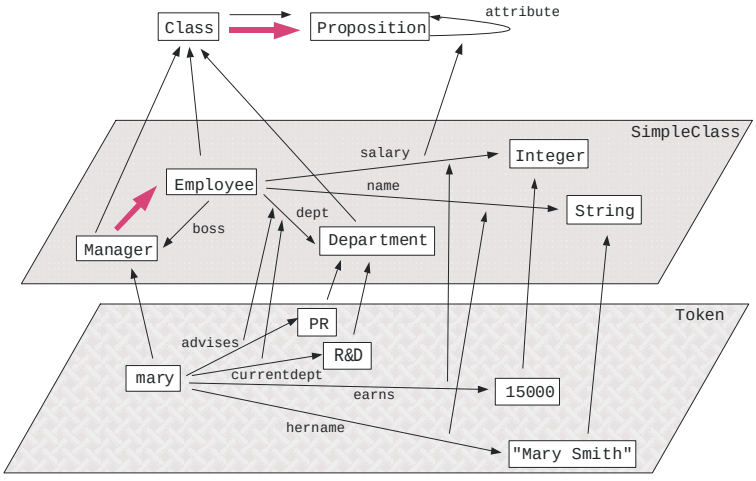

Figure 2.1: Graph representing the example proposition network

Individual objects are denoted as nodes of the graph, instantiation, specialization, and attribute relationships are represented as dotted, shaded, and labelled directed arcs between their source and destination components.

O-Telos propositions have a temporal component: the belief time[6]. The belief time of a proposition is not assigned by the user but by the system at the transaction time of an update (TELL or UNTELL). ConceptBase uses right-open and closed predefined time intervals. Right open time intervals are represented like in the subsequent example:

```
P(#mary,#mary,mary,#mary,tt(millisecond(1992,1,11,17,5,42,102),
                            infinity))
```

The object `mary` is believed since 17:05:42 on January 11, 1992. The label 'infinity' denotes that the end time of the object lies in the future and is not yet known. In any case, the current time 'Now' is regarded to be smaller than 'infinity'. Right-open belief times indicate objects that are part of the "current" knowledge base.

Closed intervals (denoted by binary tt-terms) indicate "historical" objects, i.e. objects that have been untold. Example:

```
P(#E1,#mary,*instanceof,#M,tt(millisecond(1992,1,11,17,5,42,0),
                              millisecond(1995,12,31,23,59,59,999))
```

The object `#E1`, i.e. the instantiation of `mary` to the class `Manager` is believed from 17:05:42 on January 11, 1992, until the end of the last millisecond of the year 1995. We call the first component of the belief time also the start time object and the second component the end time. Start and end time of an object can be retrieved by the predicates `Known`, and `Terminated` (see section 2.2).

2.2 Rules and constraints

The ConceptBase predicative language CBL [JK90] is used to express integrity constraints, deductive rules and queries. The variables inside the formulas have to be quantified and assigned to a "type" that limits the range of possible instantiations to the set of instances of a class. ConceptBase offers a set of predicates for the predicative language defined on top of an O-Telos object base as visible for a given rollback time:

$\mathtt{OB}_{\mathtt{rbt}} = \{\mathtt{P(oid,x,l,y)} \mid \mathtt{P(oid,x,l,y,tt)} \in \mathtt{OB}, \mathtt{rbt}$ during $\mathtt{tt}\}$

The value of the rollback time depends on the kind of formula to be processed: integrity constraints are evaluated on the current object base $\mathtt{OB}_{\mathtt{Now}}$[7] (`Now`=the smallest time interval that contains the current time). The rollback time of queries is usually provided together with the query when it is submitted from a user interface to a ConceptBase server. By default, it is `Now` as well.

2.2.1 Basic predicates

The following predicates provide the basic access to an O-Telos object base. Some have both an infix and a prefix notation. As usual we employ the object identifer to refer to an object.

1. `(x in c)` or `In(x,c)`
   The object `x` is an instance of class `c`.

2. `(c isA d)` or `Isa(c,d)`
   The object `c` is a specialization (subclass) of `d`

3. `(x m y)` or `A(x,m,y)`
   The object `x` has an attribution link to the object `y` and this link has the attribute category `m`. Structural integrity demands that the label `m` belongs to an attribute of a class of `x`.

[6]The original specification of Telos has two temporal components. The *valid time* and the *belief time*. The valid time is defined as the time interval when the statemment made by a Telos proposition is true in the world. The belief time is the time when this statement is part of the knowledge base. O-Telos skipped the valid time because it is virtually impossible to have a tractable implementation when in interplays with deductive rules. Earlier versions of ConceptBase until version 3.1 did however implement both time components.

[7]Only objects that have a right-open belief time shall be visible $\mathtt{OB}_{\mathtt{Now}}$. This is due to the fact that the end time of an object can only be changed once, namely when the object is untold. The UNTELL operation can only have happened in the past when $\mathtt{OB}_{\mathtt{Now}}$ is built.

4. `Ai(x,m,o)`
   The object `x` has an explicit attribute `o`. This attribute is instance of an attribute category with label `m`.

5. `(x m/n y)` or `AL(x,m,n,y)`
   The object `x` has an attribution link labelled `n` to the object `y`. The attribution has the category `m`.

6. `From(p,x)`
   The object `p` has source `x`.

7. `To(p,y)`
   The object `p` has destination `y`.

8. `Label(p,l)`
   The object `p` has label `l`. If `l` is used as a variable, it must be quantified over the class `Label`.

9. `P(p,x,l,y)`
   There is an object `P(p,x,l,y)` in the object base $\mathtt{OB}_{\mathtt{rbt}}$.

10. `Known(p,t)` The object `p` is known in $\mathtt{OB}_{\mathtt{rbt}}$ since `t`, i.e. an object `P(p,x,l,y,tt)` is part of the object base OB and `t` is the start time of `tt`. The argument `t` is a string of the format `"tt(millisecond(yr,mo,d,h,min,sec,millisec))"`. It is regarded as an instance of the class `TransactionTime`.

11. `Terminated(p,t)` The object `p` is unknown in $\mathtt{OB}_{\mathtt{rbt}}$ after `t`, i.e. an object `P(p,x,l,y,tt)` is part of the object base OB and `t` is the end time of `tt`. The argument `t` is represented like with `Known`. An object that has not yet been untold has the end time `"tt(infinity)"`.

12. `(x [in] mc)` or `In2(x,mc)`
    The object `x` is an instance of class `c` and `c` is an instance of class `mc`. In other words, `(x [in] mc)` is equivalent to `exists c/VAR (x in c) and (c in mc)`

13. `(x [m] y)` or `A2(x,m,y)`
    The object `x` and `y` are linked by an attribute `a1`. The attribute `a1` is an instance of an attribute `a2` which itself is an instance of an attribute `a3` with label `m`. The predicate is equivalent to the formula `exists c,d,n/VAR (x in c) and (y in d) and (c m/n d) and (x n y).`

The predicates `In2` and `A2` are also called *macro predicates* since they are standing for sub-formulas. They are fully supported in constraints of query classes. The predicate `A2` is not yet supported for deductive rules and integrity constraints due to limitations of the formula compiler. You can use the `AL` predicate instead. Examples on using macro predicates are available from the CB-Forum (`http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/877047`).

The relation of the above predicates and the P-facts of the object base is defined by the O-Telos axioms (appendix B). For example, axiom 7 states

$$\forall\, o,x,n,y,p,c,m,d\; P(o,x,n,y) \wedge P(p,c,m,d) \wedge \mathit{In}(o,p) \Rightarrow AL(x,m,n,y)$$

So, if an attribute object `o` of an object `x` is an instance of an attribute object `p` of the object `c`, then `AL(x,m,l,y)` (also written as `(x m/n y)` can be derived. This axiom provides those solutions to the `AL` predicate that are directly based on P-facts. Further solution can be derived via user-defined deductive rules. The other predicates are based on P-facts as well. The `Ai` predicate is for historical reasons not included in the list of axioms. It is defined as

$$\forall\, o,x,n,y,p,c,m,d\; P(o,x,n,y) \wedge P(p,c,m,d) \wedge \mathit{In}(o,p) \Rightarrow Ai(x,m,o)$$

There are a few variants for the predicates for instantiation, specialization and attribution to check whether a fact is actually stored or deduced:

1. `In_s(x,c)`
   The object `x` is an explicit instance of class `c`.

2. `In_e(x,c)` or `:(x in c):`
   The object `x` is an explicit instance of class `c`, or of one of the sub-classes of `c`, or of the system class of `x`. The system class of individual objects is `Individual`, attribution objects have the system class `Attribute`, instantiation objects the system class `InstanceOf`, and specialization objects have the system class `IsA`.

3. `A_e(x,m,y)` or `:(x m y):`
   The objects `x` and `y` are linked by an explict attribute with attribute category `m`. The attribute category is either explicitly assigned to the attribute or derived by a rule (see subsection 2.2.3).

4. `Isa_e(c,d)` or `:(c isA d):`
   The class `c` is a direct subclass of class `d`.

The above predicates can be used, for example, to define defaults values (see http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2396075) Since deduction should be transparent to the user, one should avoid using the above predicates if the proper predicates `In(x,c)` and `A(x,n,y)` can do the job.

2.2.2 Notes on attribution

The attribution of objects in O-Telos is more generic than in object-oriented approaches, in particular UML. In O-Telos, an attribution relates two arbitrary objects. In languages such as UML, attributes are defined at classes to declare which states an object (instance of the class) may have. This is well possible in O-Telos as well, e.g. by declaring the integer-valued `salary` attribute of a class `Employee` and using it for instances of the class. However, O-Telos does not restrict attributes to just values. The target of an attribute can be any object. Hence, the concept of an attribute in O-Telos is the generalization of an UML association and an UML attribute. A second difference is that an O-Telos attribute has essentially several labels tagged to it: its own label (object label) and the labels of its attribute categories (class labels). The latter are the labels of the attributes declared at the classes of an object, the first is the label of the attribution at the level of the object that has the attribute. We illustrate this subsequently.

Attributes at the instance level are instances of attributes at the class level (=attribute categories). An attribute category at the class level can be instantiated several times at the instance level. For example, consider the frame for `Mary`:

```
mary in Manager with
  name,aliasname
    hername: "Mary Smith"
  salary
    earns: 15000
  dept
    advises:PR;
    currentdept:RD
end
```

The object `mary` has four attributes with object labels `hername`, `earns`, `advises`, and `currentdept`. The attribute categories are `name`, `aliasname`, `salary`, and `dept`. The last category is instantiated twice. ConceptBase uses the following predicates to express the content of the frame:

```
(mary in Manager)
(mary name/hername "Mary Smith")
(mary aliasname/hername "Mary Smith")
(mary salary/earns 15000)
(mary dept/advises PR)
(mary dept/currentdept RD)
```

So, there are four attributes using four attribute categories. Like an object can have multiple classes, an attribute can have multiple categories. In fact, explicit attributes in O-Telos are just objects and their attribute categories are their classes. At the lowest abstraction level (tokens), the object labels of the attributions frequently do not carry a specific meaning and can then be neglected when formulating logical expressions. The attribution predicate `(x m y)` performs just this projection. In the example, the following attributions would be true:

```
(mary name "Mary Smith")
(mary aliasname "Mary Smith")
(mary salary 15000)
(mary dept PR)
(mary dept RD)
```

The class labels `name`, `aliasname` etc. are defined at an abstraction level where the meaning of some application domain is captured. The class label (attribute category) of an attribute is defined as an object label of an attribute at the class level. For example, the `name` and `aliasname` attributes could be defined for the class `Employee` as follows:

```
Employee in Class with
   attribute,single
      name: String
   attribute
      aliasname: String
end
```

Here, the following predicate facts would be true:

```
(Employee in Class)
(Employee attribute/name String)
(Employee single/name String)
(Employee attribute/aliasname String)
(Employee attribute String)
(Employee single String)
```

The mechanism for attribution is exactly the same as for instances like `mary`. Note that the 3-argument attribution predicate expressing `(mary name "Mary Smith")` represents a meaningful statement for some reality to be modeled. On the other hand, the predicate fact `(Employee attribute String)` is much less significant because the label `attribute` does not transport a specific domain meaning. Here, the 4-argument attribution predicate such as used for the fact `(Employee attribute/name String)` is required. Still, from a formal point of view, there is no different treatment of predicates at the class and instance level. This uniformity is the basis for meta-modeling, i.e. the definition of modeling languages by means of meta classes. The class labels `attribute` and `single` need to be defined at the classes of `Employee`. Those are `Class` and the pre-defined class `Proposition`, to which any object including `Employee` and `mary` is instantiated. In this case, both `attribute` and `single` are defined for `Proposition`:

```
(Proposition attribute/attribute Proposition)
(Proposition attribute/single Proposition)
```

Note that `attribute` has itself as category. This is the most generic attribute category and applies to any (explicit) attribution.

Both the attribution predicate `(x m y)` and its long form `(x m/n y)` can be derived, i.e. occur as conclusion of a deductive rule. In such cases, there are no explicit attribute objects between `x` and `y`. ConceptBase demands, that in such cases one of the classes of `x` has an attribute with label m. Deductive rules for `(x m/n y)` are introduced with ConceptBase V7.1. They allow to simulate multi-sets, i.e. derived attributes where the same value can occur multiple times. The feature is still experimental. Examples are available in the CB-Forum (http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2330042).

2.2.3 Assigning attribute categories to explicit attributes

The instantiation of an explicit attribute to an attribute category can be explicit (see above), or via inheritance, or via a user-defined rule. Explicit instantiation is typically established when telling a frame like the `Employee` example to the database. *Instantiation by inheritance* is more rarely used but is in fact just the application of the specialization principle to attribution objects:

```
Employee with
   attribute
     salary: Integer
end
Manager isA Employee with
   attribute
      bonus: Integer
end
Manager!bonus isA Employee!salary end
```

Here, the `bonus` attribute is declared as specialization of the `salary` attribute. Any instance of the `bonus` attribute will then be an instance of the `salary` attribute via the usual class membership inheritance of O-Telos. For example,

```
mary in Manager with
   bonus
     bon1: 10000
end
```

shall make the following attribution facts true:

```
(mary bonus/bon1 10000)
(mary bonus 10000), A_e(mary,bonus,10000)
(mary salary/bon1 10000)
(mary salary 10000), A_e(mary,salary,10000)
(mary attribute/bon1 10000)
(mary attribute 10000), A_e(mary,attribute,10000)
```

The third method to instantiate an explicit attribute to an attribute category is via a *user-defined rule*. We use the employee example again:

```
Employee in Class with
   attribute
     salary: Integer;
     premium: Integer;
     country: String
   rule
     premrule: $ forall e/Employee prem/Employee!premium
                    (e country "NL") and Ai(e,premium,prem)
                  ==> (prem in Employee!salary) $
end
```

Now consider the following instances:

```
marijke in Employee with
  salary sal: 50000
  premium pr: 3000
  country ctr: "NL"
end
```

This would make the following attribution facts true:

```
(marijke salary/sal 50000)
(marijke salary 50000), A_e(marijke,salary,50000)
(marijke premium/pr 3000)
(marijke premium 3000), A_e(marijke,premium,3000)
(marijke salary/pr 3000)
(marijke salary 3000), A_e(marijke,salary,3000)
(marijke country/ctr "NL")
(marijke country "NL"), A_e(marijke,country,"NL")
```

Hence, any explicit premium attribute of an employee of the Netherlands is regarded as an explicit salary as well.

Note that the three cases discussed here are for *explicit* attribution objects. You may also define rules that derive `(x m y)` or `(x m/n y)` directly. In such cases, there is no need for an explicit attribute between `x` and `y`. The attribution is complelety derived.

2.2.4 Reserved words

In order to avoid ambiguity, neither `in` and `isa` nor the logical connectives `and` and `or` are allowed as attribute labels[8]. Likewise, names of predicates such as `A`, `Ai`, `In` should not be used as object names or variable names. The same holds for the keywords `with` and `end`, which are used in the frame syntax.

2.2.5 Comparison predicates

The next predicates are second class citizens in formulas. In contrast to the above predicates they cannot be assigned to classes of the O-Telos object base base. Consequently, they may only be used for testing, i.e. in a legal formula their parameters must be bound by one of the predicates 1 - 8.

1. `(x < y), (x > y), (x <= y), (x >= y)`
   `x` and `y` may be instances of any class. If they are instance of `Integer` or `Real`, they are ordered numerically. If they are instance of `TransactionTime` they are ordered according to the time they are representing (newer times are greater than older times). Otherwise, they are ordered alphabetically.

2. `(x = y)`
   The objects `x` and `y` are equal.

3. `(x <> y)` or `(x \= y)`
   The objects `x` and `y` are not the same.

All comparison predicates may use functional expressions as operands. They are evaluated before the comparison predicates is evaluated. See section 2.3.3 for examples. The predicates `(x == y)`, `UNIFIES(x,y)` and `IDENTICAL(x,y)` defined in earlier releases of ConceptBase are deprecated. It is recommended to use `(x = y)` instead.

2.2.6 Typed variables

The exact syntax of CBL is given in appendix A. The types of variables (i.e. quantified identifiers) are interpreted as instantiations:

- `forall x/C F` → `forall x (x in C) ==> F`

[8]For the example of subsection 2.1 among others the ground predicates `(mary in Manager)`, `(Manager isA Employee)` and `(mary earns 15000)` are valid facts describing the contents of the object base. We suggest to choose verbs (e.g. `earns` in our example) for attribute labels to get more natural and readable predicates.

- `exists x/C F` → `exists x (x in C) and F`

The class `C` attached to variable `x` is called the *variable range*. The anonymous variable range `VAR` is treated as follows.

- `forall x/VAR F` → `forall x F`
- `exists x/VAR F` → `exists x F`

Anonymous variable ranges are only permitted in meta formulas, see section 2.2.9.

2.2.7 Semantic restrictions on formulas

We demand that each variable is quantified exactly once inside a formula. This is no real restriction: in case of double quantification rename one of the variables. More important is a restriction similar to static type checking in programming languages that demands a strong relationship between formulas and the knowledge base:

Predicate typing condition

> (1) Each constant (= arguments that are not variables) in a formula F must be the name of an existing object in the O-Telos object base, or it is a constant of the builtin classes Integer, Real, or String.
>
> (2) For each attribution predicate `(x m y)` (or `Ai(x,m,o)`, resp.) occuring in a formula there must be a unique attribute labelled `m` of some class `c` of `x` in the knowledge base, the so-called *concerned class*.
>
> (3) For each instantiation predicate `(x in c)`, the argument `c` must be a constant.

All instantiation and attribution predicates need to be "typed" according to the predicate typing condition. Formally, we don't assign types to such predicates but *concerned classes*. Any instantiation predicate and any attribution predicate in a formula must have a unique concerned class. It is determined as follows:

- The concerned class of an instantiation predicate `(x in c)` is the class `c`. The argument `c` may not be a variable.
- The concerned class of attribution predicates `(x m y)` and `Ai(x,m,o)` is principally the most special attribute with label `m` of all classes of `x`[9]. The O-Telos axioms listed in appendix B, in particular axiom 17, make sure that there may not be more than one candidate attribute if `x` is the name of an existing object. If `x` is a variable, we demand that there is at most one candidate in the variable range of `x` and its superclasses. If no class of x (i.e. also no superclass of any class of x) defines such an attribute and the CBserver has been started with the predicate typing mode 'extended', then the concerned class is determined from the subclasses of the classes of x. Theoretically, one can choose the common superclass of all such attributes of subclasses of the classes of x (if existent). However, ConceptBase currently demands that there must be a single such attribute in the subclass hierarchy.

Example: The concerned class of `(e boss b)` in the `SalaryBound` constraint in subsection 2.2.8 is the `Employee!boss`. The class of variable `e` is `Employee`. This is the most special superclass of itself and indeed defines the attribute `Employee!boss`.

The purpose of the predicate typing condition is to allow ConceptBase to compile attribution predicates `(x m y)` to an internal form `Adot(cc,x,y)` that replaces the attribute label `m` by the object identifier `cc` of the concerned class. This enourmously speeds up the computation of predicate extensions. A similar effect is applicable to instantiation predicates. Here, the concerned class of `(x in c)` is `c`. Another effect of the predicate typing condition is that certain semantically meaningless predicate occurrences are

[9] Since any object is an instance of `Proposition`, ConceptBase will include this class when searching the concerned class of an attribution predicate.

detected at compile time. For example, `(x m y)` can only have a non-empty extension, if some class of `x` defines an attribute with label `m`.

If the argument `x` in a predicate `(x m y)` is a variable, then the initial class of `x` is determined by the the variable range in the formula. The variable `this` of query class constraints can have multiple initial classes, being the set of superclasses of the corresponding query class. All superclasses of `c` are also regarded as classes of `x`. If `x` is a constant, then the classes of `x` are determined by a query to the object base. A formula violating the first clause of the predicate typing condition would make a statement about something that is not part of the object base. As an example, consider the following formula:

```
forall x/Emplye not (x boss Mary)
```

With the example object base of section 2.1, we find two errors: There are no objects with names `Emplye` and `Mary`.

There are two possible cases to violate the second part of the restriction. The first case is illustrated by an example:

```
forall x/Proposition y/Integer (x salary y) ==> (y < 10000)
```

In this case the classes of x, Proposition and any of its superclasses, have no attribute labelled salary. Therefore, the predicate (x salary y) cannot be assigned to an attribute of the object base. Instead, one has to specify

```
forall x/Employee y/Integer (x salary y) ==> (y < 10000)
```

or

```
forall x/Manager y/Integer (x salary y) ==> (y < 10000)
```

depending on whether the formula applies to managers or to all employees.

The second clause of the predicate typing condition is closely related to multiple generalization/instantiation. Suppose, we add new classes `Shop`, `Guest` and `GuestEmployee` to the given class `Employee`:

```
Shop in Class
end

Guest in Class with
      attribute
            dept: Shop
end

GuestEmployee in Class isA Guest,Employee
end
```

The following formula refers to objects of class `GuestEmployee` and their dept attribute. The problem is that two different attributes, `Employee!dept` and `Guest!dept`, apply as candidates for the predicate `(x dept PR)`:

```
forall x/GuestEmployee (x dept PR) ==> not (x in Manager)
```

In order to solve this ambiguity, we demand that in such cases a common subclass exists that defines an attribute dept which conforms to both definitions, e.g.

```
Shop in Class
end

GuestEmployee with
        attribute
                dept: ShopDepartment
end

ShopDepartment in Class isA Shop,Department
end
```

The third clause of the predicate typing condition is forbidding instantiation predicates with a variable in the class postion. The restriction is a pre-condition for an efficient implementation of the incremental

formula evaluator of ConceptBase. Without a constant in the class position of `(x in c)` any update of the instances of any class matches the predicate. Hence, ConceptBase would need to re-evaluate the formula that contains the predicate. Since any update (TELL,UNTELL,RETELL) is containing instantiation facts, any formula with an unrestricted predicate `(x in c)` has to be re-evaluated for any update. This inefficiency can be avoied by demanding that the class position is a constant. A relaxation to this clause (and clause 2) is discussed in sub-section 2.2.9.

When compiling the frames, ConceptBase will make sure that the attribute `GuestEmployee!dept` is specializing the two `dept` attributes of `Shop` and `Department`. As a consequence, the attribution predicate `(x dept PR)` can be uniquely attached to its so-called *concerned class* `GuestEmployee!dept`.

The predicate typing condition holds for all formulas, regardless whether they occur as constraints or rules of classes or within query classes[10].

2.2.8 Rule and constraint syntax

A legal *integrity constraint* is a CBL formula that fulfills predicate typing condition. A legal *deductive rule* is a CBL formula fulfilling the same condition and having the format:

`forall x1/c1 ... forall xn/cn R ==> lit(a1,...,am)`

where

- `lit` is a predicate of type 1 or 3, and
- the variables in `a1,...,am` are contained in `x1,...,xn`

In O-Telos, rules and constraints are defined as attributes of classes. Use the category `constraint` for integrity constraints, and the category `rule` for deductive rules. The text of the formula has to be enclosed by the character '`$`'. The choice of the class for a rule or constraint is arbitrary (except for query classes which use the special variable 'this').

Continuing our running example, the following formula is a deductive rule that defines the `boss` of an `Employee`. Note that the variables `e,m` are `forall`-quantified.

```
Employee with
 rule
   BossRule : $ forall e/Employee m/Manager
                (exists d/Department
                (e dept d) and (d head m))
                ==> (e boss m) $
 constraint
   SalaryBound : $ forall e/Employee b/Manager x,y/Integer
                (e boss b) and (e salary x) and (b salary y)
                   ==> (x <= y) $
end
```

The second formula is an integrity constraint that uses the boss attribute defined by the above rule. The constraint demands a `salary` of an `Employee` does not exceed the `salary` of his `boss`. Note that you can define multiple salaries for a given instance of `Employee`. The constraint is on each individual `salary`, not on the sum[11]! Also note that the arguments of the <= predicate are bound by the two predicates with attribute label `salary`.

[10] The enforcement of the restriction has been extended to query classes as of ConceptBase release 6.1. To support applications that were written for earlier releases, a CBserver option `-cc` (predicate typing) has been introduced to disable the check for query classes. Details are in section 6. With ConceptBase 7.2 (March 2010), the predicate typing has been further extended and now will scan subclasses of of the classes of x in attribution predicates (x m y) in case that superclasses do not provide a matching attribute class. You need to set the CBserver option `-cc` to 'extended' to activate this behavior. The extended mode creates more cases that the predicate type (=concerned class) is found. It should be noted that objects like x that are not instance of a class that defines and attribute with label m will lead to a failure of the predicate (x m y), i.e. its negation is then true.

[11] Use multi-sets as discussed in `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2330042` if you want to constrain the sum of salaries.

2.2.9 Meta formulas

Some formulas violating the predicate typing condition can be re-written to a set of formulas that do not violate the condition. The so-called *meta formulas* are a prominent category of such formulas. They have occurrences of `(x in c)` and/or `(x m y)` where `c` or `m` are variables (also called *meta variables*. In such cases, the concerned class cannot be determined directly even though the formula as such is meaningful. ConceptBase relies on predicate typing for the sake of efficiency and static stratification. The concerned class is internally used as predicate name. This increases the selectivity and reduces the chance on non-stratified deduction rules. Fortunately, all meta formulas can be re-written to formulas fulfilling the predicate typing condition. The re-writing replaces the meta variables by all possible value. Since all variables are bound to finite classes, the re-writing yields a finite set of formulas. However, if a meta variable is bound to a class with a large extension, the re-writing will also yield a large set of generated formulas.

Meta formulas allow to specify assertions involving objects from different levels and hence significantly improve flexibility of O-Telos models. An example for the usage of meta formulas can be found in the appendix D.2 where the enforcement of constraints in ER diagrams is solved in an elegant way.

As instructional example, assume we want to define that a certain attribute category `M` is transitive, i.e. if `(x M y)` and `(y M z)`, then `(x M z)` shall hold. Many attribute categories are supposed to be transitive, for example the `ancestor` relation of persons, or the `connection` relation between cities in a railway network.

The following meta formula defines transitivity once and forever:

```
Proposition in Class with
  attribute
     transitive: Proposition
  rule
     trans_R:
     $ forall x,y,z,M/VAR
          AC/Proposition!transitive
          C/Proposition
           P(AC,C,M,C) and (x in C) and
           (y in C) and (z in C) and
           (x M y) and (y M z) ==> (x M z) $
end
```

The rule is a meta formula because `C` and `M` are meta variables. In this case, one can re-write the formula by replacing all possible fillers for `AC`, i.e. by the instances of `Proposition!transitive`. A filler for `AC` will determine fillers for `C` and `M` since the first argument of a proposition `P(AC,C,M,C)` is identifying the proposition.

As a consequence, one can define the ancestor relation to be transitive by simply telling

```
Person in Proposition with
  transitive
     ancestor: Person
end
```

ConceptBase will match the attribute `Person!ancestor` with the variable `AC` in the above meta formula. This yields `P(Person!ancestor,Person,ancestor,Person)`, which binds the meta variable `C` to `Person` and `M` to `ancestor`. The resulting generated formula is:

```
forall x,y,z/VAR  (x in Person) and (y in Person) and (z in Person)
        and (x ancestor y) and (y ancestor z)
                    ==> (x ancestor z)
```

which can be shortened to

```
forall x,y,z/Person (x ancestor y) and (y ancestor z)
                    ==> (x ancestor z)
```

This formula is fulfilling the predicate typing condition. Likewise, the connection relation of cities gets transitive via:

```
City in Proposition with
  transitive
     connection: City
end
```

The advantage of meta formulas is that they save coding effort by re-using them in different modelling contexts. If a meta formula is linked to an attribute category (like `transitive` in the example above, then the semantic of several such attribute category can be combined by just specifying that a certain attribute has multiple categories. Assume for example that we have defined acyclicy with a similar meta formula:

```
Proposition in Class with
  attribute
     acyclic: Proposition
  constraint
     acyclic_IC:
     $ forall x,y,M/VAR
              AC/Proposition!acyclic
              C/Proposition
                P(AC,C,M,C) and (x in C) and
                 (y in C) and
                 (x M y)  ==> not (y M x) $
end
```

Then, the `ancestor` attribute can be specified to be both transitive and acyclic by

```
Person in Proposition with
  transitive,acyclic
     ancestor: Person
end
```

The more categories like `transitive` and `acyclic` are defined with meta formulas, the greater is the productivity gain for the modeler. Not only does it save coding effort. It also reduces coding errors since formula specification is a difficult task. Meta formulas are a natural extension to classical meta modeling. They allow to specify the meaning of modeling constructs at the meta class level. The mapping to simple formulas allows an efficient evaluation. It also allows to retrieve the specialized semantics definition of a model (instance of a meta model) since the generated simple formulas are attached to the constructs of the model (in the example above they are attached to classes `Person` and `City`). The meta formula compiler is fully incremental, i.e. if the object base is updated, then the set of generated simple formulas is also updated if necessary. For example, if one removes the category `transitive` from the `connection` attribute of `City`, then the generated simple formula will also be removed.

Meta formulas that contain meta variables under existential quantification cannot be compiled directly, but there is an elegant trick to circumvene this restriction. Consider for example the formula:

```
$ forall x/VAR SC/CLASS spec/ISA_complete
     (spec super SC) and (x in SC) ==>
      exists SUBC/CLASS (spec sub SUBC) and (x in SUBC) $
```

The meta variable `SUBC` is under an existential quantifier. To circumvene the problem, we write an intermediary rule replacing the predicate `(x in SUBC)`:

```
$ forall x/Proposition spec/ISA SUBC/CLASS
     (spec sub SUBC) and (x in SUBC) ==> (x inSubRel SUBC) $
```

and then re-write the original constraint to

```
$ forall x/VAR SC/CLASS spec/ISA_complete
     (spec super SC) and (x in SC) ==>
     exists SUBC/CLASS (spec sub SUBC) and (x inSubRel SUBC) $
```

So essentially, we pass the meta variable to the condition of the intermediary rule. The attribute `inSubRel` is just used to be able to specify a dedicated conclusion predicate for the intermediary deductive rule. It is defined as attribute of `Proposition`. The complete example is at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/d3070600/mp-ISA-complete.sml.txt.

Many more re-usable examples for meta formulas are in the ConceptBase-Forum at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1042523.

2.2.10 Further object references

In addition to the so called *select expressions* `!`, `=>`, `->` already introduced above for directly refering to attributes, specializations and instantiations as objects, three other basic constructors may be used within frames and assertions.

- `^` is the counterpart of `!` and denotes the target of an attribute instead of the attribute object itself, e.g. `mary^advises` is the same as `PR`.
- The set valued `.` operator has the commonly used meaning as in paths in object-oriented models and relates an object with the set of all attribute values of a certain category, i.e. `mary.dept` contains both `PR` and `R&D`.
- As `.` can be understood as the set variant of `^` but employing the attribute *category* instead of the concrete attribute label, the same holds for `|` (with respect to `!`). Thus `mary|dept` is the set of all attributes (as objects) that belong to category `dept` and have source `mary`.

Note, that `.` and `|` are only allowed to occur within assertions whereever classes may be interpreted as range restrictions, e.g. in quantifications or at the right hand side of `in` predicates. The full syntax which allows combinations of all basic constructors can be found in the appendix. For illustration we just give two examples here. The first is an alternative representation for the rule above, the second could be a constraint stating that all bosses of `Mary` earn exactly 50.000.

1. `forall e/Employee m/Manager (m in e.dept.head) ==> (e boss m).`
2. `forall b/Mary.dept.head (b salary 50000)`

2.2.11 User-definable error messages for integrity constraints

ConceptBase provides a couple of errors messages in case of an integrity violation. These errors messages refer to the logical definition of the constraint and are sometimes hard to read. To provide more readable error messages, one can attach so-called *hints* to constraint definitions. These hints are attached as comments with label `hint` to the attribute that defines the constraint.

Consider the salary bound constraint above. A hint could look like:

```
Employee!SalaryBound with
 comment
   hint: "An employee may not earn more than her/his manager!"
end
```

It is also possible to attach hints to meta-level constraints. In this case, the hint text can refer to the meta-level variables occuring in the meta-level constraint. These variables will be replaced by the correct fillers when the meta-level constraint is utilized in some modeling context.

Assume, for example, we want to have a symmetry category and attach a readable hint to it:

```
Proposition with
  attribute
    symmetric: Proposition
end

RelationSemantics in Class with
  constraint
 symm_IC: $ forall AC/Proposition!symmetric C/Proposition x,y/VAR M/VAR
                     P(AC,C,M,C) and (x in C) and (y in C) and
                     (x M y)  ==> (y M x) $
end

RelationSemantics!symm_IC with
 comment
   hint: "The relation {M} of {C} must be symmetric,
          i.e. (x {M} y) implies (y {M} x)."
end
```

Note that the references to the *meta variables*[12] `M` and `C` are surrounded by curly braces, and that these meta variables are also occurring in the meta-level constraint. Now, use the `symmetric` concept in some modeling context, e.g. to define that the `marriedTo` attribute of `Person` should be symmetric:

```
Person with
  attribute,symmetric
   marriedTo: Person
end
```

At this point of time, ConceptBase will find the hint text for the symmetric constraint and will adapt it to the context of `C=Person` and `M=marriedTo`. When an integrity violation occurs, the *substituted* hint

```
"The relation marriedTo of Person must be symmetric,
 i.e. (x marriedTo y) implies (y marriedTo x)."
```

will be presented to the user. An example violation is:

```
bill in Person with
  marriedTo m1: eve
end

eve in Person end
```

One can also define a hint for the meta-level constraint that refers only to a (non-empty) subset of the meta variables. If a hint for a meta formula cannot be substituted as shown avove, ConceptBase will not issue the hint but rather the text of the generated formula.

Examples of user-defined error messages can be found in the ConceptBase-Forum at `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1543277`.

2.3 Query classes

ConceptBase realizes queries as so-called query classes, whose instances fulfill the membership constraint of the query [Stau90]. This section first defines the structural properties of the query language CBQL and then introduces the predicative component. Queries are instances of a system class `QueryClass` which is defined as follows:

[12] Meta variables are those variables that occur in the class position of (x in c), or in the label position of (x m y) or Label(x,m). In the running example, C and M are meta variables.

```
QueryClass in Class isA Class with
     attribute
         retrieved_attribute: Proposition;
         computed_attribute: Proposition
     attribute,single
         constraint: MSFOLquery
end
```

A super classes of query class imposes a range condition of the set of possible instances of the query class: any instance of the query class must be an instance of the superclass. Example: "socially interested" are those managers that are member of a union.

```
Union in Class
end

UnionMember in Class with
    attribute
        union:Union
end

SI_Manager_0 in QueryClass isA Manager,UnionMember
end

QueryClass SI_Manager in QueryClass isA Manager,UnionMember with
     retrieved_attribute
         union: Union;
         salary: Integer
end
```

Super classes themselves may be query classes, which is the first kind of query recombination. The second frame shows the feature of **retrieved attributes** which is similar to projection in relational algebra. Example: one wants to see the name of the union and the salary of socially interested managers. The attributes must be present in one of the super-classes of the query class. In this example, the `union` attribute is obviously inherited from the class `UnionMember` and `salary` is inherited from `Manager`. CBQL demands that retrieved attributes are necessary: each answer must instantiate them. If an object does not have such an attribute then it will not be part of the solution. As usual with attribute inheritance, one may specialize the attribute value class, e.g.

```
QueryClass Well_off_SI_Manager isA SI_Manager with
     retrieved_attribute
         salary: HighSalary
end

HighSalary in Class isA Integer with
     rule
       highsalaryrule: $ forall m/Integer
                           (m >= 60000)
                                 ==> (m in HighSalary) $
end
```

The new attribute value class `HighSalary` is a subclass of `Integer` so that each solution of the restricted query class is also a solution of the more general one. It should also be noted that `HighSalary` also could have been another query class. This is the second way of query recombination.

Retrieved attributes and super-classes already offer a simple way of querying a knowledge base: projection and set intersection. For more expressive queries there is an predicative extension, the so-called **query constraint**. We use the same many-sorted predicative language as in section 2.2 for deductive rules and integrity constraints and introduce a useful abbreviation:

Let Q be a query class with a constraint F that contains the predefined variable this. Then, the query class is essentially an abbreviation for the two deduction rules

```
forall this F' ==> Q(this)
forall this Q(this) ==> (this in Q)
```

The deduction rules are generated by the query compiler and only listed here for discussing the meaning of a query class. The variable this stands for any answer object of Q. We call this also the answer variable. The sub-formula F' is combined from the query constraint F and the structural properties of the query, in particular the super-classes and the retrieved attributes. Each super-class C of Q contributes a condition (this in C) to the sub-formula F'. Each retrieved attribute like a:D contributes a condition ((this a v) and (v in D)) to F'. Moreover, each retrieved attribute add the new argument v to the predicate Q. The following example shows the translation.

```
QueryClass Well_off_SI_Manager1 isA SI_Manager with
     retrieved_attribute
         union: Union
     constraint
         well_off_rule: $ exists s/HighSalary
                        (this salary s) $
end
```

The generated deduction rules for this query class are:

```
forall this,v
   (this in SI_Manager) and
   (this union v) and (v in Union) and
   (exists s (s in HighSalary) and (this salary s))
==> Well_off_SI_Manager1(this,v)

forall this,v Well_off_SI_Manager1(this,v)
==> (this in Well_off_SI_Manager1)
```

Classes occuring in a query constraint may be query classes themselves, e.g. HighSalary. This is the third way of query recombination.

The next feature introduces so-called **computed attributes**, i.e. attributes that are defined for the query class itself but not for its super-classes. The assignment of values for the solution is defined within the query constraint. Like retrieved attributes, computed attributes are included in the answer predicate so that the proper answer can be generated from it.

The following example defines a computed attribute head_of that stands for the department a manager is leading. The attribute head_of is supposed to be computed by the query. It is not an attribute of SI_Manager or its super-classes. We expect that and answer to the query includes the computed attribute. Note that a reference ~head_of to the computed attribute occurs inside the query constraint.

```
QueryClass Well_off_SI_Manager2 isA SI_Manager with
     retrieved_attribute
         union: Union
     computed_attribute
         head_of: Department
     constraint
         well_off_rule: $ exists s/HighSalary
                    (this salary s) and
                    (~head_of head this) $
end
```

The variable ~head_of in the constraint is prefixed with ~ to indicate that it is a placeholder for the compted attribute with label head_of. We recommend to use the prefix to avoid confusion of the placeholder variable in query constraints and corresponding attribute label in the query definitions. Analogously, you can use the prefixed answer variable ~this instead of the plain version this. ConceptBase will accept both the prefixed and the non-prefixed version for the answer variable and the placeholder variable

of computed attributes. Non-prefixed placeholders in constraints are replaced internally by the prefixed counterparts.

The generated deduction rules for above query would be:

```
forall this,v1,v2
   (this in SI_Manager) and
   (this union v1) and (v1 in Union) and
   (v2 in Department) and
   (exists s (s in HighSalary) and (this salary s)
              and (v2 head this))
==> Well_off_SI_Manager2(this,v1,v2)

forall this,v1,v2 Well_off_SI_Manager2(this,v1,v2)
==> (this in Well_off_SI_Manager2)
```

Computed attributes are treated differently from retrieved attributes. The retrieved attribute `union` causes the inclusion of the condition `(this union v1) and (v1 in Union)`. The corresponding variable `v1` does not occur in the sub-formula generated for the query class constraint. The computed attribute causes the inclusion of the condition `(v2 in Department)` but typically also occurs in the query constraint. Like retrieved attributes computed attributes are necessary, i.e. any solution of a query with a computed attribute must assign a value for this attribute. There is no limit in the number of retrieved and computed attributes. The more of them are defined for a query class, the more arguments shall the answer predicate have.

Recursion can be introduced to queries by using recursive deductive rules or by refering recursively to query classes. The example asks for all direct or indirect bosses of `bill`:

```
QueryClass BillsMetaBoss isA Manager with
     constraint
         billsBosses:
            $ (bill boss this) or
              exists m/Manager
                (m in BillsMetaBoss) and
                (m boss this)$
end
```

Further examples can be found in the directory

`$CB_HOME/examples/QUERIES`.

Queries are represented as O-Telos classes and consequently they can be stored in the knowledge base for future use. It is a common case that one knows at design time **generic queries** that are executed at run-time with certain parameters. CBQL supports such parameterizable queries:

```
GenericQueryClass isA QueryClass with
     attribute
        parameter: Proposition
end
```

Generic queries are queries of their own right: they can be evaluated. Their speciality is that one can easily derive specialized forms of them by substituting or specializing the parameters. An important property is that each solution of a substituted or spezialized form is also a solution of the generic query. This is a similar to the inheritance paradigm. The example shows that parameters can be retrieved or computed attributes at the same time: (note, that variable for the parameter in the constraint is prefixed here with ~; you may also omit the prefix in the constraint as explained above).

```
What_SI_Manager in GenericQueryClass isA Manager,UnionMember with
     retrieved_attribute,parameter
        salary: HighSalary;
```

```
        u: Union
     computed_attribute,parameter
        head_of: Department
     constraint
        well_off_rule: $(this union ~u) and (~head_of head this)$
end
```

There are two kinds of specializing generic query classes:

1. Specialization of a parameter `[a:C']`

   Example: `What_SI_Manager[salary:TopSalary]`

   In this case `TopSalary` must be a subclass of `HighSalary`. The solutions are those managers in `What_SI_Manager` that not only have a high but a top salary.

2. Instantiation of a parameter `[v/a]`

   Example: `What_SI_Manager[Research/head_of]`

   The variable `head_of` is the replaced by the constant `Research` (which must be an instance of `Department`).

One may also combine several specializations, e.g.

```
What_SI_Manager[salary:TopSalary,Research/head_of].
```

The specialized queries can occur in other queries in any place where ordinary classes can occur, e.g.

```
QueryClass FavoriteDepartment isA Department with
     retrieved_attribute
         head: What_SI_Manager[10000/salary]
end
```

Parameters that don't occur as computed or retrieved attributes are interpreted as existential quantifications if they are not instantiated.

2.3.1 Query definitions versus query calls

Telling a frame that declares an instance of `QueryClass` (as well as its sub-classes `GenericQueryClass`, and `Function`) constitutes the definition of a query. It shall be compiled internally into Datalog code not visible to the user. Once defined, a query can be called simply by referring to its name. Hence, if `Q` is a the name of a defined query class, then `Q` is also an admissable query call. It results in the set of all objects that fulfill the membership constraint of `Q`. ConceptBase regards these objects as derived instances of the query class `Q`.

If a query class has parameters, then any of its specialized forms is also an admissable query call. For example, if `Q` has two parameters `p1`, `p2` in its defining frame, then `Q[v1/p1,v2/p2]` is the name of a class whose instances is the subset of instances of `Q` where the parameters `p1` and `p2` are substituted by the values `v1` and `v2`. The substitution yields a simplified membership constraint that precisely defines the extension of `Q[v1/p1,v2/p2]`.

If a generic query class is called with all parameters substituted by fillers, then one can omit the parameter labels. Assume that the query `Q` has just the parameters `p1` and `p2`. Then the expression `Q[v1,v2]` is equivalent to `Q[v1/p1,v2/p2]`. ConceptBase uses the alphabetic order of paremeter labels to convert the shortcut form to the full form.

2.3.2 Query classes and deductive integrity checking

ConceptBase regards query classes as ordinary classes with the only exception that class membership cannot be postulated (via a TELL) but is derived via the class membership constraint formulated for the query class. A consequence of this equal treatment is that a constraint formulated for an ordinary class can refer directly or indirectly to a query class, e.g.:

```
Unit in Class with
  attribute
    sub: Unit
end
BaseUnit in QueryClass isA Unit with
  constraint
    c1: $ not exists s/Unit!sub From(s,~this) $
end
SimpleUnit in Class isA Unit with
  constraint
    c: $ forall s/SimpleUnit (s in BaseUnit) $
end
```

Here, the constraint in the class `SimpleUnit` refers to the query class `BaseUnit`.

ConceptBase supports references to query classes without parameters[13] in ordinary class constraints and rules. A prerequisite is that the the query class is an instance of the builtin class `MSFOLrule`. Membership to this builtin class is necessary to store the generated code for an integrity constraint (or a rule that an integrity constraint might depend upon) and to enable the creation of a dependency network between the query class and the integrity constraints. There are two simple methods to achieve membership to `MSFOLrule`.

Method 1: Make sure that any query class is an instance of `MSFOLrule`. This can simply be achieved be telling the following frame prior to your model:

```
QueryClass isA MSFOLrule end
```

Method 2: Decide for each query class individually. You tag only those query classes that are used in rules or constraints. This individual treatment saves some code generation at the expense of being less uniform. Such an individual tagging would look like

```
BaseUnit in QueryClass,MSFOLrule isA Unit with
  constraint
    c1: $ not exists s/Unit!sub From(s,~this) $
end
```

ConceptBase will reject an integrity constraint or rule if it refers to a query class that is not an instance of `MSFOLrule`.

If a query class is defined as instance of `MSFOLrule`, then it should not have a meta formula as constraint! This is a technical restriction that can easily be circumvened by using normal deductive rule.

For example, instead of the query class

```
UnitInstance in QueryClass,MSFOLrule isA Proposition with
  constraint
     c1: $ (~this [in] Unit) $
end
```

you should define

[13] If the CBserver option `-cc` is set to off, we also allow calls to generic query classes in rules and constraints. In such cases, incremental integrity checking will be incomplete and thus potentially wrong. Only experienced users should employ them.

```
UnitInstance in Class with
  rule
     r1: $ forall x/VAR (x [in] Unit) ==> (x in UnitInstance) $
end
```

The example uses the macro predicate `(x [in] Unit)` explained earlier in this section. It is equivalent to the sub-formula `exists c (x in c) and (c in Unit)`.

2.3.3 Nested query calls and shortcuts

ConceptBase has capabilities to form nested expressions from generic query classes. The idea is to combine them like nested functional expressions, e.g. $f(g(x), h(y))$. The problem is however that queries stand for predicates and nested query calls are thus formally higher-order logic (predicates occur as arguments of other predicates), and consequently outside Datalog. Still, the feature is so useful that we provide it. A nested query call is like an ordinary parameterized query call except that the parameters can themselves be query calls. For example, `COUNT[What_SI_Manager[10000/salary]/class]` counts the instances of the parameterized query call `What_SI_Manager[10000/salary]`. Syntactically, query calls can be arbitrarily deep, e.g.

```
Union[Intersec[EmpMinSal[800/minsal]/X,
               EmpMaxSal[1400/maxsal]/Y]/X,
      Manager/Y]
```

ConceptBase does perform the usual type check on the parameters by analyzing the instantiation of the *core class* of a query call. For example, the core class of `EmpMinSal[800/minsal]` is `EmpMinSal`. Thus, ConceptBase will check whether `EmpMinSal` is an instance of the class expected for the parameter `X`.

Nested query calls are mostly used in combination with functional expressions, i.e. nested query calls where queries are functions (see section 2.5). A function in ConceptBase is a query class that has at most one answer object for any combination of input parameters. Of particular interest for nested query calls are functions that do not operate on values (suchs as integers) but rather on classes such as the class of all employees with more than two co-workers. ConceptBase provides a collection of aggregate functions that operate on classes. For example, the `COUNT` function returns the number of instances of a class. The input of such a function can be any nested query call.

```
QueryClass EmployeeWith2RichCoworkers isA Employee with
  constraint
    c2: $ (COUNT[RichCoworker[this/worker]/class] = 2) $
end
```

The outer predicate (here: `COUNT`) is an instance of `Function`, i.e. delivers at most one value for the given argument. It is also possible that both operands of a comparison predicate are nested expressions:

```
QueryClass EmployeeWithMoreRichCoworkersThanWilli isA Employee with
  constraint
    c2: $ (COUNT[RichCoworker[this/worker]/class] >
           COUNT[RichCoworker[Willi/worker]/class]) $
end
```

ConceptBase supports shortcuts for query calls and function calls (see section 2.5) in case that all parameters of a query (or function) have fillers in the query call. In such cases, one can write `Q[x1,x2,...]` instead of `Q[x1/p2,x2/p2,...]`. ConceptBase shall replace the actual parameters `x1,x2` etc. for the parameter labels `p1,p2` in the *alphabetic order* of the parameter labels. For example, the expression `RichCoworker[this]` is equivalent to `RichCoworker[this/worker]` since `worker` is the only parameter label of the query. Likewise, `COUNT[c]` is a shortcut for `COUNT[c/class]`. Since `COUNT` is also a function, we support `COUNT(c)` as well to match the usual notation for function expressions. The last query class is thus equivalent to:

```
QueryClass EmployeeWithMoreRichCoworkersThanWilli isA Employee with
  constraint
    c2: $ (COUNT(RichCoworker[this]) > COUNT(RichCoworker[Willi])) $
end
```

Since the `COUNT` function is frequently used, ConceptBase provides the shortcut `#c` for `COUNT(c)`. Consequently, the shortest form of the above query would be:

```
QueryClass EmployeeWithMoreRichCoworkersThanWilli isA Employee with
  constraint
    c2: $ (#RichCoworker[this] > #RichCoworker[Willi]) $
end
```

The shortcut is also applicable to the `Union` example above. The expression below computes the numbers of instances of the set expression.

```
#Union[Intersec[EmpMinSal[800],EmpMaxSal[1400]],Manager]
```

The definitions for `Union` and `Intersec` can be found in the ConceptBase-Forum at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/896920.

Besides `COUNT`, ConceptBase supports aggregation function for finding the minimum, maximum and average of a set. Aggregation functions are not limitered to numerical domains. For example, one can define a function that returns an arbitrary instance of a class:

```
selectrnd in Function isA Proposition with
  parameter
    class: Proposition
end
```

The membership constraint has to be provided as program code, see chapter E. A call

```
selectrnd(RichCoworker[Willi/worker])
```

would then return an arbitrary instance of `RichCoworker[Willi/worker]`. Random functions can be useful in the context of active rules (section 4), e.g. to initiate the firing of a rule with an arbitrary candidate out of the set of candidates. The code for `selectrnd` is accessible via the ConceptBase-Forum at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1694234.

2.3.4 Reified query calls

You might want to memorize certain query calls that you want to call over and over again. ConceptBase provides a built-in class `QueryCall`, which you can instantiate by such query calls as ordinary objects, i.e. *reified query calls*. The following example defines the class count as a query call object:

```
COUNT[Class/class] in QueryCall end
```

Of course, you can ask the query call `COUNT[Class/class]` without having told it as an object. Reifying `COUNT[Class/class]` additionally allows you to use it as an attribute of another object, or to browse it with the graph editor. Examples for query calls, in particular for using integer intervals as class attributes, are available in the CB-Forum at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2571997.

2.4 View definitions

The view language of ConceptBase is an extension of the ConceptBase Query Language CBQL. Besides some extensions that allow an easier definition of queries, views can also be nested to express n-ary relationships between objects.

The system class `View` is defined as follows:

```
Class View isA GenericQueryClass with
attribute
    inherited_attribute : Proposition;
    partof : SubView
end
```

Attributes of the category `inherited_attribute` are similar to retrieved attributes of query classes, but they are not necessary for answer objects of the views, i.e. an object is not required to have a filler for the inherited attribute for being in the answer set of the view.

The `partof` attribute allows the definition of complex nested views, i.e. attribute values are not only simple object names, they can also represent complex objects with attributes. The following view retrieves all employees with their departments, and attaches the head attribute to the departments.

```
View EmpDept isA Employee with
retrieved_attribute, partof
    dept : Department with
        retrieved_attribute
            head : Manager
    end
end
```

As the example shows, the definition of a complex view is straightforward: for the "inner" frame the same syntax is used as for the outer frames. The answers of this view are represented in the same way, e.g.

```
John in EmpDept with
dept
    JohnsDept : Production with
        head
            ProdHead : Phil
    end
end

Max in EmpDept with
dept
    MaxsDept : Research with
        head
            ResHead : Mary
    end
end
```

To make the definition of views easier, we allow some shortcuts in the view definition for the classes of attributes.

For example, if you want all employees who work in the same departments as `John`, you can use the term `John.dept` instead of `Department`. In general, the term `object.attrcat` refers to the set of attribute values of `object` in the attribute category `attrcat`. This path expressions may be extended to any length, e.g. `John.dept.head` refers to all managers of departments in which John is working.

A second shortcut is the explicit enumeration of allowed attribute values. The following view retrieves all employees, who work in the same department as John, and earn 10000, 15000 or 20000 Euro.

```
View EmpDept2 isA Employee with
retrieved_attribute
```

```
    dept : John.dept;
    salary : [10000,15000,20000]
end
```

As mentioned before, "inner" frames use the same syntax as normal frames. You can also specify constraints in inner frames which refer to the object of an outer frame.

```
View EmpDept_likes_head isA Employee with
retrieved_attribute,partof
    dept : Department with
        retrieved_attribute, partof
            head : Manager with
                constraint c : $ A(this,likes,this::dept::head) $
        end
    end
end
```

The rule for using the variable "this" in nested views is, that it always refers the object of the main view, in this case an employee. Objects of the nested views can be referred by `this::label` where `label` is the corresponding attribute name of the nested view. In the example, we want to express that the employees must like their bosses. Because the inner view for managers is already part of the nested view for departments we must use the double colon twice: `this::dept` refers to the departments and `this::dept::head` refers to the managers.

If you reload the definition of a view into the Telos Editor, the complex structure of it is lost. During compilation of the view, the view is translated into several classes and some additional contraints are generated, so the resulting objects might look quite strange if you reload them.

2.5 Functions

Functions are special queries which have some input parameters and return at most one object as result for one set of input parameters. Functions can either be user-defined by a membership constraint like for regular query classes, or they may be implemented by a PROLOG code, which is defined either in the OB.builtin file (this file is part of every ConceptBase database) or in a LPI-file (see also section 4.2.2).

A couple of aggregation functions are predefined for counting, summing up, and computing the minimum/maximum/average. Furthermore, there are functions for arithmetic and string manipulation. See section E.2 for the complete list. Since functions are defined as regular Telos objects, you can load their definition on the Telos editor of the ConceptBase User Interface.

Unlike as for user-defined generic query classes, you have to provide fillers for all parameters of a function. We will refer to any query expression whose outer-most query is a function as a *functional expression*.

The intrinsic property of a function is that it returns at most one return object[14] This property allows to form complex functional expressions including arithmetic expressions. Functions are also regarded as special query classes. Subsequently, we introduce first how to define and use functions like queries. Then, we define the syntax of functional expressions and the definition of recursive functions such as the computation of the length of the shortest path between two nodes.

2.5.1 Functions as special queries

Assume that an attribute (either explicit or derived) has at most one filler. For example, a class `Project` may have attributes `budget` and `managedBy` that both are single-valued. A third attribute `projMember` is multi-valued.

[14] A function returns the empty set *nil* if it is undefined for the provided input values.

```
Project with
  attribute,single
    budget: Integer;
    managedBy: Employee
  attribute
    projMember: Employee
end
```

The two functions `ProjectBudget` and ProjectManager return the corresponding objects:

```
ProjectBudget in Function isA Integer with
  parameter
    proj: Project
  constraint
    c1: $ (proj budget this) $
end

ProjectManager in Function isA Employee with
  parameter
    proj: Project
  constraint
    c1: $ (proj managedBy this) $
end
```

The two functions share all capabilities of query classes, except that the parameters are required (one cannot call a function without providing fillers for all parameters) and that there is at most one return object per input value.

Function can be called just like queries, for example `ProjectBudget[P1/proj]` shall return the project budget of project P1 (if existent). You can also use the shortcut `ProjectBudget[P1]` like for any other query.

2.5.2 Shortcuts for function calls and functional expressions

Since functions require fillers for all parameters, ConceptBase offers also the standard syntax `f(x)` to refer to function calls. The expression `f(x)` is a shortcut for `f[x/param1]`, where `param1` is the only parameter of function `f`. If a function has more than one parameter, then they are replaced according their alphabetic order:

```
g in Function isA T with
  parameter
    x: T1;
    y: T2
  constraint
    ...
end
```

A call like `g(bill,1000)` is a shortcut for `g[bill/x,1000/y]` because `x` is occurs before `y` in the ASCII alphabet. The shortcut can be used to form complex functional expressions such as `f(g(bill,ProjectBudget(P1)))`. There is no limitation in nesting function calls. Function calls are only allowed as left or right side of a comparison operator. They are always evaluated before the comparison operator is evaluated. For example, the equality operator

```
$  ... (f(g(bill,ProjectBudget(P1))) = f(g(mary,1000))) $
```

will be evaluated by evaluating first the inner functions and then the outer functions. Note that the parameters must be compliant with the parameter definitions.

As a special case of functional expressions, ConceptBase supports arithmetic expressions in infix syntax. The operator symbols +, −, ⋆, and / are defined both for integer and real values. ConceptBase shall determine the type of a sub-expression to deduce whether the to use the real-valued or integer-valued variant of the operation. Examples of admissable arithmetic expressions are

```
a+2*(b-15)
n+f(m)/3
```

Provided that `a` and `b` are integers, the first expresssion is equivalent to the function shortcut

```
IPLUS(a,IMULT(2,IMINUS(b,15)))
```

and to the query call

```
IPLUS[a/i1,IMULT[2/i1,IMINUS[b/i1,15/i2]/i2]/i2]
```

The second arithmetic expression includes a division which in general results in a real number. Hence, the function shortcut for this expression is

```
PLUS(n,DIV(f(m),3))
```

The whole expression returns a real number.

2.5.3 Example function calls and definitions

1. The following three variants all count the number of instances of `Class`:

   ```
   COUNT[Class/class]
   COUNT(Class)
   #Class
   ```

   The result is an integer number, e.g.

   ```
   119
   ```

   The operator # is a special shortcut for `COUNT`.

2. The subsequent query sums up the salaries of an Employee:

   ```
   SUM_Attribute[bill/objname,Employee!salary/attrcat]
   SUM_Attribute(Employee!salary,bill)
   ```

   Note that the parameter label `attrcat` is sorted before `objname` for the function shortcut. The result is returned as a real number, even if the input numbers were integers.

   ```
   2.5001000000000e+04
   ```

   You can also use functions also in query class to assign a value to a “computed_attribute”:

   ```
   QueryClass EmployeesWithSumSalaries isA Employee with
   computed_attribute
       sumsalary : Real
   constraint
       c: $ (sumsalary = SUM_Attribute(Employee!salary,this)) $
   end
   ```

3. Complex computations can be made by using multiple functions in a row. This query returns the percentage of query classes wrt. the total number of classes.

```
Function PercentageOfQueryClasses isA Real with
constraint
    c: $ exists i1,i2/Integer r/Real
    (i1 = COUNT[QueryClass/class]) and (i2 = COUNT[Class/class]) and
    (r = DIV[i1/r1,i2/r2]) and (this = MULT[100/r1,r/r2]) $
end
```

The query can be simplified with the use of function shortcuts to

```
Function PercentageOfQueryClasses isA Real with
constraint
    c: $ (this = MULT(100,DIV(COUNT(QueryClass),
                              COUNT(Class)))) $
end
```

and with arithmetic expressions to

```
Function PercentageOfQueryClasses isA Real with
constraint
    c: $ (this = 100 * #QueryClass / #Class) $
end
```

The function `PercentageOfQueryClasses` has zero parameters. You can use it as follows in logical expressions

```
$ ... (PercentageOfQueryClasses() > 25.5) ... $
```

So, a function call `F()` calls a query `F` that has no parameter.

Functions that yield a single numerical value can directly be incorporated in comparison predicates. For example, the following query will return all individual objects that have more than two attributes:

```
ObjectWithMoreThanTwoAttributes in QueryClass isA Individual with
attribute,constraint
    c1 : $ (COUNT_Attribute(Proposition!attribute,this) > 2) $
end
```

The functional expression used in the comparison can be nested. See section 2.3.3 for details. You can also re-use the above query to form further functional expressions, e.g. for counting the number of objects that have more than two attribute. You find below all three representations for the expression.

```
COUNT[ObjectWithMoreThanTwoAttributes/class]
COUNT(ObjectWithMoreThanTwoAttributes)
#ObjectWithMoreThanTwoAttributes
```

2.5.4 Programmed functions

If your application demands functional expressions beyond the set of predefined-functions, you can extend the capabilities of your ConceptBase installation by adding more functions. There are two ways: first, you can extend the capabilities of a certain ConceptBase database, or secondly, you can add the new functionality to your ConceptBase system files. We will discuss the first option in more details using the function `sin` as an example, and then give some hints on how to achieve the second option.

A function (like any builtin query class) has two aspects. First, the ConceptBase server requires a regular Telos definition of the function declaring its name and parameters. This can look like:

```
sin in Function isA Real with
  parameter
    x : Real
  comment
    c : "computes the trigonometric function sin(x)"
end
```

The super-class `Real` is the range of the function. i.e. any result is declared to be an instance of `Real`. The parameters are listed like for any regular generic query class. The comment is optional. We recommend to use short names to simplify the constructions of functional expressions. The above Telos frame must be permanently stored in any ConceptBase database that is supposed to use the new function.

The second aspect of a function is its implementation. The implementation can be in principal in any programming language but we only support PROLOG because it can be incrementally addded to a ConceptBase database. An implementation in another programming language would require a re-compilation of the ConceptBase server source code. The syntax of the PROLOG code must be conformant to the Prolog compiler with which ConceptBase was compiled. This is in all cases SWI-Prolog (www.swi-prolog.org). For our `sin` example, the PROLOG code would look like:

```
compute_sin(_res,_x,_C) :-
        cbserver:arg2val(_x,_v),
        number(_v),
        _vres is sin(_v),
        cbserver:val2arg(_vres,_res).

tell: 'sin in Function isA Real with parameter x : Real end'.
```

The first argument `_res` is reserved for the result. then, for each parameter of the function there are two arguments. The first is for the input parameter (`_x`), the second holds the identifier of the class of the parameter (here: `_C`). It has to be included for technical reasons. The clause 'tell:' instructs ConceptBase to tell the Telos definition when the LPI file is loaded. Instead of this clause you may also tell the frame manually via the ConceptBase user interface.

There are a few ConceptBase procedures in the body of the `compute_sin` that are of importance here. The procedure `cbserver:arg2val` converts the input parameter to a Prolog value. ConceptBase internally always uses object identifiers. They have to be converted to the Prolog representation in order to enter them into some computation. The reverse procedure is `cbserver:val2arg`. It converts a Prolog value (e.g. a number) into an object identifier that represents the value. If necessary, a new object is created for holding the new value.

The above code should be stored in a file like `sin.swi.lpi`. This file has to be copied into the ConceptBase database which holds the Telos definition of `sin`. You will have to restart the ConceptBase server after you have copied the LPI file into the directory of the ConceptBase database.

If you want the new function to be available for all databases you construct, then you have to copy the file `sin.swi.lpi` to the subdirectory `lib/system` of your ConceptBase installation. Note that your code might be incompatible with future ConceptBase releases. If you think that your code is of general interest, you can share it with other ConceptBase users in the Software section of the CB-Forum (http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2768063).

2.5.5 Recursive function definitions

Some functions like the Fibonacci numbers are defined recursively. ConceptBase supports such recursive definitions. If the function is defined in terms of itself, then you just express the recursive definition in the membership constraint of the function:

```
fib in Function isA Integer with
  parameter
    n: Integer
  constraint
```

```
    cfib: $ (n=0) and (this=0) or
            (n=1) and (this=1) or
            (n>1) and (this=fib(n-1)+fib(n-2))
          $
end
```

ConceptBase shall internally compile the disjunction into three formulas:

```
forall n,this/Integer (n=0) and (this=0) ==> fib(this)
forall n,this/Integer (n=1) and (this=1) ==> fib(this)
forall n,this/Integer (n>1) and (this = fib(n-1)+fib(n-2)) ==> fib(this)
```

ConceptBase employs the magic cache query evaluator to evaluate the recursive function. Thus, the result of a function call `fib(n)` shall only be computed once and then re-used in subsequent calls.

If the recursion is not inside a single function definition but rather a property of a set of function/query definitions, then you must use so-called *forward declarations*. They declare the signature of a function/query before it is actually defined. A good example is the computation of the length of the shortest path between two nodes.

```
spSet in GenericQueryClass isA Integer with
  parameter
    x: Node;
    y: Node
end
sp in Function isA Integer with
  parameter
    x: Node;
    y: Node
  constraint
    csp: $ (x=y) and (this=0) or
           (x nexttrans y) and (this = MIN(spSet[x,y])+1)
         $
end
spSet in GenericQueryClass isA Integer with
  parameter
    x: Node;
    y: Node
  constraint
   csps: $ exists x1/Node (x next x1) and (this=sp(x1,y)) $
end
```

Here, the query class `spSet` computes the set of length of shortest path between the successors of a node `x` and a node `y`. The length of the shortest path is then simply 0 if `x=y` or the minimum of the `spSet[x,y]` plus 1 if there is a path from x to y, and undefined else. The signature of `spSet` must be known for compiling `sp` and vice versa. ConceptBase has a single-pass compiler. Hence, it requires the forward declaration. The query `spSet` is not a function because it returns in general several numbers. The complete example for computing the length of the shortest path is in the CB-Forum, see http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1694234.

Recursive function definitions require much care. Deductive rules shall always return a result after a finite computation. This does not hold in general for recursive function definitions when they use arithmetic subexpressions. These subexpressions can create new objects (numbers) on the fly and thus force ConceotBase into an infinite computation. On the other hand, they are very expressive and useful to analyze large models in a quantitative way.

2.6 Query evaluation strategies

ConceptBase employs an SLDNF-style query evaluation method, i.e. query predicates are evaluated top-down much like in standard Prolog. This is known to cause infinite loops for certain recursive rule sets.

To overcome this, the SLDNF evaluator is augmented by a *caching sub-system*, which detects recursive predicate calls and answers them from the cached results of a query rather than entering an infinite loop. This cache-based evaluation computes the fixpoint (=answer) of a query provided that the overall rule set is stratified. Even more, dynamically stratified rule sets are supported as well. Other than with the static stratification test, a violation is detected at run time of a query rather than at compile time.

For a precise definition of stratification, we refer you to the literature on deductive databases. For the purposes of this manual, consider the following rule:

```
forall p/Position (exists p1/Position (p moveTo p1) and not (p1 in Win))
  ==> (p in Win)
```

ConceptBase internally compiles such rules into a representation where `Position`, `moveTo`, and `Win` are predicate symbols:

```
forall p
   (exists p1 Position(p) and Position(p1) and moveTo(p,p1) and not Win(p1))
  ==> Win(p)
```

Static stratification requires that one can consistently assign stratification levels (=numbers) to the set of predicate symbols such that

1. If there is a rule with conclusion predicate A and positive condiction predicate B (=not negated), then the level of A must be greater or equal the level of B.

2. If there is a rule with conclusion predicate A and negated condiction predicate B , then the level of A must be strictly greater the level of B.

In the example above, the conclusion predicate `Win` depends on the condition predicate `not Win`. Since we only can assign one level to `Win`, we cannot find a static stratification for the above rule. The same argument also works in case of multiple inter-dependent rules. Static stratification can be tested at compile-time of a rule.

Dynamic stratification is an extension of static stratification, i.e. any statically stratified rule set is also dynamically stratified. It is not only considering predicate symbols but also the arguments with which a predicate is called at run-time. Obviously, this depends on the database state at a certain point of time. The global rule of dynamic stratification is that the answer to a predicate call `A(x)` may not depend on its negation `not A(x)`. Such a clash can be detected by maintaining a stack of active predicate calls.

ConceptBase reports a violation of dynamic stratification in the log window of the CB client with a message "STRATIFICATION VIOLATION" indicating the predicate that participated in the violation. Two flavors of error messages can occur:

- **STRATIFICATION_VIOLATION1** The evaluation of a predicate P leads to a cyclic call to itself, where in between a negated predicate $\neg Q$ occurs. Hence, P depends on $\neg Q$ and Q depends on P. Consequently, P indirectly depends on its negation.

- **STRATIFICATION_VIOLATION2** While evaluating a predicate $\neg P(a)$ the system derives that $P(a)$ would be true, i.e. the fact $P(a)$ is in the extension of the variable free predicate call $P(a)$. This evaluation occurs in the context of a matching positive call $P(x)$ where the system cannot prove that $P(a)$ would be in the extension. The matching call $P(x)$ has the same arguments as $P(a)$ except for certain variable positions.

In practice, most rule sets are already statically stratified, i.e. no violation can occur regardless of the data in the database. Counter examples are in the CB-Forum (see `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/888832`) in the models Russel.sml and Win.sml. These examples are neither statically nor dynamically stratified. Note also the example WinNim.sml which uses the same query as Win.sml but is dynamically stratified. Even in the case of stratification violations, ConceptBase will display an answer to a query. The user can then decide which parts of the answer are usable. In principle, this decision can also be automated which is subject of future ConceptBase releases[15].

[15] Older releases of ConceptBase provided a prototype of a query evaluation strategy based on the magic set method. We no longer support this strategy.

2.7 Datalog queries and rules

The definition of queries in ConceptBase is often complicated by the limitations of the expressiveness of the query language or by the limitations of the query optimizer to find the best solution. The concept of datalog queries and rules was introduced to overcome these limitations. Datalog queries and rules give the experienced user the possibility to define the executable code of query (or rule) directly, including the use of standard PROLOG predicates such as ground/1 to improve the performance of a query or rule.

Although datalog queries and rules can be used as any other query (or rule), they can cause an inconsistent database. This is due to the fact, that the datalog queries and rules will usually not be evaluated while the semantic integrity of the database is checked.

2.7.1 Extended query model

Datalog queries are defined in a similar way as standard query classes. They must be declared as instance of the class `DatalogQueryClass`.

```
Class DatalogQueryClass isA GenericQueryClass with
attribute
   code : String
end
```

The attribute `code` defines the executable code of the query as string.

Datalog rules have to be defined as an instance of `DatalogInRule` or `DatalogAttrRule`, depending on whether their conclusion should be an In-predicate or an A-predicate.

```
Class DatalogRule with
attribute
  concernedClass : Proposition;
  code : String
end

Class DatalogInRule isA DatalogRule
end

Class DatalogAttrRule isA DatalogRule
end
```

The attribute `concernedClass` specifies the class for the In-predicate or the attribute class for the A-predicate.

2.7.2 Datalog code

The datalog code is a list of predicates, separated by commas (,). As in Datalog or Prolog, this will be interpreted as a conjunction of the predicates. To use disjunction, the `code` attribute has to be specified multiple times.

All predicates that may be used in standard rules and queries may also be used in datalog queries (see section 2.2 for a list). An argument of a predicate may be one the following:

- an object name: If the object name starts with an upper case letter or includes special characters such as `!` or `"`, it must be written in single quotes (`'`). This also holds for string object, e.g. `"a string"` must be written as `'"a string"'`.
- a predefined variable, defined by the context of the query or rule: If the query has a super class, the variable `this` refers to the instances of this class. If the query has parameters or computed attributes, variables with the names of the parameters or computed attributes will be predefined.

In a DatalogInRule, the variable this refers to the instances of the concerned class. In a DatalogAttrRule, the variables src and dst refer to the source and destination object of the attribute. Note, that **all predefined variables have to be prefixed with ~ and must be encoded in single quotes (')**, e.g. '~this', '~src', '~param'.

- existential variables: These variables must be declared in a special predicate vars(list), which has to be the first predicate of the code. For example, the predicate vars([x,y]) defines the variables x and y.

A query expression of the form query(q) may be also used as predicate, or as second argument of an In-predicate. q may be any valid query expression, e.g. just the name of a query class, or a derive expression including the specification of parameters (for example, find_instances[Class/class]).

In addition, PROLOG predicates can be used as predicates. You can define your own PROLOG predicates in a LPI-file (see section 4.2.2 for an example).

2.7.3 Examples

This section defines a few datalog queries and rules for the standard example model of Employees, Departments and Managers (see $CB_HOME/examples/QUERIES).

The first example defines a more efficient version of the recursive MetabossQuery.

```
DatalogQueryClass MetabossDatalogQuery isA Manager with
  attribute,parameter
     e : Employee
  attribute,code
     r1 : "In('~e','Employee'),A('~e',boss,'~this')";
     r2 : "vars([m]),
           In('~e','Employee'),
           In(m,query('MetabossDatalogQuery[~e/e]')),
           A(m,boss,'~this')"
end
```

Note that the disjunction of the original query is represented by two code-attributes. The example shows also the use of query expressions and existential variables.

The second example is the datalog version of the rule for the HighSalary class. The infix-predicate >= is represented by the predicate GE.

```
Class HighSalary2 isA Integer
end

DatalogInRule HighSalaryRule2 with
concernedClass
   cc: HighSalary2
code
   c: "In('~this','Integer'),GE('~this',60000)"
end
```

The last example shows the definition of a rule for an attribute. It also shows, how the performance of a rule can be improved by specifying different variants for different binding patterns. The example defines two rules, depending on the binding of the variable src. The rule defines the transitive closure of the boss attribute. Rule r1 is applied, if both arguments src and dst are unbound. The second rule is used, if at least src is bound, and the last rule will be applied, if we have a binding for dst but not for src.

```
DatalogAttrRule MetabossRule with
concernedClass
```

```
     cc : Employee!boss
code
    r1 : "vars([m]),var('~src'),var('~dst'),In(m,'Manager'),
          A(m,boss,'~dst'),A('~src',boss,m)";
    r2 : "vars([m]),ground('~src'),
          A('~src',boss,m),A(m,boss,'~dst')";
    r3 : "vars([m]),ground('~dst'),var('~src'),
          A(m,boss,'~dst'),A('~src',boss,m)"
end
```

Note, that the predicates `var` and `ground` are builtin predicates of PROLOG. Thus, this is also an example for calling PROLOG predicates in a query or rule.

Chapter 3

Answer Formats for Queries

3.1 Basic definitions

By default, ConceptBase displays answers to queries in the FRAME format (see 'A' and 'B' below). For many applications, other answer representations are more useful. For example, relational data is more readable in a table structure. Another important example are XML data. If ConceptBase is integrated into a Web-based information system, then answers in HTML format are quite useful. For this reason, answer format definitions are provided.

Answer formats in ConceptBase are based on term substitutions where terms are evaluated against substitution rules defined by the answers to a query. A substitution rule has the form $L \longrightarrow R$ with the intended meaning that a substring L in a string is replaced by the substring R. The object of a term substitution is a string in which terms may occur, for example:

this is a string called `{a}` with a term `{b}`

Assume the substitution rules:

- `{a}` $\longrightarrow$ string no. `{x}`
- `{b}` $\longrightarrow$ that was subject to substitution
- `{x}` $\longrightarrow$ 123

The derivation of a string with terms proceeds from left to right. First, the term occurence `{a}` is dealt with. The next term in the string is then `{x}` which is evaluated to `123`. Finally, `{b}` is substituted and the result string is `this is a string called string no. 123 with a term that was subject to substitution.`

We denote a single derivation step of a string S_1 to a string S_2 by $S_1 \Longrightarrow S_2$. It is defined when there occurs a substring L in S_1, i.e. $S_1 = V + L + W$ and a substitution rule $L \longrightarrow R$ and $S_2 = V + R + W$. The substrings V and W may be empty. A string S is called ground when no substition rule can be applied. A sequence $S \Longrightarrow S_1 \Longrightarrow \ldots \Longrightarrow S_n$ is called a derivation of S. A complete derivation of S ends with a ground string. In our example, the complete derivation is:

```
      this is a string called {a} with a term {b}.
⟹    this is a string called string no. {x} with a term {b}.
⟹    this is a string called string no. 123 with a term that was
      subject to substitution.
```

An exception to the left-to-right rule are complex terms like `{do({y})}`. Here, the inner term `{y}` is first evaluated (e.g. to 20) and then the result `{do(20)}` is evaluated.

In general, term substitution can result in infinite loops. This looping can be prevented either by restricting the structure of the substitution rule or by terminating the substitution process after a finite number of steps. The end result of a substitution process of a string is called its derivation. In ConceptBase, the

substitution rules are guaranteeing termination except for the case of external procedures. The problem with the exception is solved by prohibiting cyclic calls of the same external procedure during the substitution of a call. A cyclic call is a call that has the same function name (e.w. query class) and the same arguments (expressed as parameter substitutions).

In ConceptBase, an answer format is an instance of the new pre-defined class 'AnswerFormat'.

```
Individual AnswerFormat in Class with
  attribute
     forQuery : QueryClass;
     order : Order;
     orderBy : String;
     head : String;
     pattern : String;
     tail : String;
     fileType: String
end
```

The first attribute assigns an answer format to a query class (a query may have at most one answer format). The second and third attribute specify the sorting order of the answer, i.e. one can specify by which field an answer is sorted, and whether the answer objects are sorted 'ascending' or 'descending' (much like in SQL).

The 'head', 'pattern', and 'tail' arguments are strings that define the substring substitution when the answer is formatted. They contain substrings of type expr that are replaced. The head and tail strings are evaluated once, independent form the answer to the query. Usually, they do not contain expressions but only text. The response to a query is a set of answer objects A1,A2,.... The pattern string is evaluated against each answer object. For each answer object, the derivation of the pattern is put into the answer text. Hence, the complete answer consists of

```
      derivation of head string
  +   derivation of pattern string for answer object A1
  +   derivation of pattern string for answer object A2
      ...
  +   derivation of tail string
```

The `fileType` attribute is explained in section 3.4.

In the next sections, we will explain more details about answer formats using the following example: An answer object A to a query class QC has by default a 'frame' structure

```
A in QC with
  cat1
     label11: v11;
     label12: v12
     [...]
  cat2
     label21: v21
  [...]
end
```

In case of a complex view definition VC, the values vij can be answer objects themselves, e.g.

```
B in VC with
  cat1
     label11: v11;
     label12: v12
     [...]
  cat2
     label21: v21 with
```

```
                cat21
                  label211: v211
              [...]
              end
    [...]
end
```

3.2 Constructs in answer formats

3.2.1 Simple expressions in patterns

We first concentrate on the pattern attribute, i.e. they are not applicable in the head or tail attribute of an answer format. The pattern of an answer format is applied to *each* answer object of a given query call, effectively transforming it according to the pattern. The following expressions are allowed. Capital letters in the list below indicate that the term is a placeholder for some label occuring in the query definition or the answer objects.

{**this**} denotes the object name of an answer object (e.g. 'A'). The syntax for object names is defined in section 2.1. In particular, attribution objects have names like `mary!earns`.

{**this.ATTRIBUTE_CAT**} denotes the value(s) of the attribute 'ATTRIBUTE_CAT' of the current answer object 'this'. The attribute must be defined in the query class (retrieved_attribute, computed_attribute) or view (inherited_attribute). Note that some attributes are multi-valued. For the answer object 'A', `{this.cat1}` evaluates to `v11,v12` and `{this.cat2}` evaluates to `v21`. For the complex object 'B', a path expression like `{this.cat2.cat21}` is allowed and yields `v21`. Note that all such expressions are set-valued.

{**this^ATTRIBUTE_LABEL**} denotes the value of the attribute 'ATTRIBUTE_LABEL' of the current answer object 'this'. The attribute_label is at the level of the answer object, i.e. not at the class level but at the instance level. Therefore, this expression is rarely used because the instance level attribute labels are usually unknown at query definition time. Example A and B: `{this^label12}` evaluates to `v12`.

{**this|ATTRIBUTE_CAT**} denotes all attribute labels of the answer object 'this' that are grouped under the category 'ATTRIBUTE_CAT' (defined in the query class). Example: `{this|cat1}` evaluates to `label11,label12`.

The derivation of pattern is performed for each answer object that is in the result set of a query. The answer object 'A' induces the following substition rules:

```
          {this}  ⟶  A
     {this.cat1}  ⟶  v11,v12
     {this.cat2}  ⟶  v21
  {this^label11}  ⟶  v11
  {this^label12}  ⟶  v12
  {this^label21}  ⟶  v21
     {this|cat1}  ⟶  label1,label2
     {this|cat2}  ⟶  label21
```

Extended examples for these simple expressions are given in `simple_answerformats1.sml` and `simple_answerformats2.sml` in the directory `$CB_HOME/examples/AnswerFormat/`.

Note that only objects that match the query constraint are in the answer. Particularly, the categories `computed_attribute` and `retrieved_attribute` require that at least one filler for the respective attribute is present in an answer object! Use ConceptBase views ('View') and `inherited_attribute` in case that zero fillers are also allowed for answers.

There are few other simple expressions that may be useful. They just list attributes without having to refer to specific attributes.

{**this.attrCategory**} denotes all attribute categories that are present in an answer object. Example: For answer object 'A' `{this.attrCategory}` evaluates to `cat1,cat2`.

{**this|attribute**} lists all attribute labels occuring in an answer object. Example: for answer object 'A', `{this|attribute}` evaluates to `label11,label12,label21`.

{**this.attribute**} lists all attribute values occuring in an answer object. Example: for answer object 'A', `{this.attribute}` evaluates to `v11,v12,v21`.

{**this.oid**} displays the internal object identifier of the answer object {this}.

For the answer object A, these expressions induce the following additional substitution rules:

```
{this.attrCategory}  ⟶  cat1,cat2
   {this|attribute}  ⟶  label11,label12,label21
   {this.attribute}  ⟶  v11,v12,v21
```

3.2.2 Pre-defined variables

The following variables can be used in the head, tail, and pattern of an answer format. They do not refer to the variable *this*.

{**user**} outputs the user name of the current transaction; typically has the structure name@address

{**transactiontime**} the time when the current transaction was started; has format YYYY/MM/DD hh:mm:ss and is based on Coordinated Universal Time (UTC), formerly known as Greenwhich Mean Time

{**cb_version**} the version number of ConceptBase

{**cb_date_of_release**} the version number of ConceptBase

{**currentmodule**} is expanded to the name of the current module.

{**currentpath**} is expanded to the complete module path (starting with root module `System`) that was active when starting the transaction

{**database**} is expanded to relative or absolute path of the database that was specified with the `-d` option of the CBserver (section 6); if no database was specified, the variable is expanded to `<none>`

ConceptBase also adds those command line parameters that deviate from their defaults to the set of pre-defined variables. The most common ones are (see also section 5.9):

{**loadDir**} : directory from which the CBserver loads Telos source files at start-up; command-line parameter `-load`

{**saveDir**} : directory into which the CBserver saves Telos sources of modules at shut-down or client logout; command-line parameter `-save`

{**viewDir**} : directory into which the CBserver materializes results of certain queries command-line parameter `-views`

Moreover, if the current transaction was a call of a query like `MyQuery[v1/param1,...]` then {param1} will be evaluated to `v1`. This makes all parameter substitutions of a query call available to answer formatting.

3.2.3 Iterations over expressions

In case of expressions with multiple values, the user may want to generate a complex text that uses one value after the other as a parameter. This is in particular useful to transform multiple attribute values like `this.cat1`. The 'Foreach' construct has the format:

`{Foreach( (expr1,expr2,...), (x1,x2,...), expr )}`

The expression `expr1` is evaluated yielding each a list of solutions `s11,s12,...` The same is applied to `expr2` yielding a list `s21,s22,...` Then, the variables `x1,x2,...` are matched against the first entries of all lists, i.e. `x1=s11,x2=s21,...` This binding is then applied to the expression `expr` which should contain occurences of `{x1}`, `{x2}`, ... This replacement is continued with the second entries in all lists yielding bindings `x1=s12,x2=s22,...` This is continued until all elements of all lists are iterated. If some lists are smaller than others, the missing entries are replaced by `NULL`.

During each iteration, the new bindings induce substitution rules for the binding

Iteration 1:

`{x1}` ⟶ `s11`
`{x2}` ⟶ `s21`
...

Iteration 2:

`{x1}` ⟶ `s12`
`{x2}` ⟶ `s22`
...

Note that the third argument `expr` may contain other subexpressions, even a nested 'Foreach'. An example for iterations is given in `$CB_HOME/examples/AnswerFormat/iterations.sml`.

The Foreach construct contains three arguments separated by commas. These two commas are used by ConceptBase to parse the arguments of the Foreach-construct and similar answer formatting expressions. They are not printed to the answer stream.

3.2.4 Special characters

If one wants a comma inside an expresssion that shall be visible in the answer, then one has to escape it '\,'. The same holds for the other special characters like '(', ')' etc. Here is a short list of supported special characters. Some require a double backslash.

`\\n`	:	new line (ASCII character 10)
`\\t`	:	tab character
`\\b`	:	backspace character
`\0`	:	empty string (no character)
`\(`	:	left parenthesis
`\)`	:	right parenthesis
`\,`	:	comma

In principal, any non-alphanumerical character like '(','),'{','}', '[', ']' can be referred to by the backslash operator. Note that the vanilla versions of these characters are used to denote expressions in answer formats. Hence, we need to 'escape' them by the backslash if they shall appear in the answer.

3.2.5 Function patterns

The substitution mechanism for answer formats recognizes patterns such as

`{F(expr1,expr2,...)}`

as function calls. An example is the `ASKquery` construct from from section 3.2.6. The mechanism is however very general and can be used to realize almost arbitrary substititions. The parentheses and the commas separating the arguments of `F` are parsed by ConceptBase and not placed on the output. The following simple function patterns are pre-defined:

- `{From(expr)}` inserts the source object of `expr`. The source object is computed by the predicate `From(x,o)` of section 2.2. This pattern is useful for printing attributes. ConceptBase will first apply pattern substitution to `expr` and then to the whole term `{From(expr)}`.
- `{To(expr)}` inserts the destination object of `expr`. The destination object is computed by the predicate `To(x,o)` of section 2.2.
- `{Label(expr)}` inserts the label of the object referenced by `expr`. The label is computed by the predicate `Label(x,l)` of section 2.2.
- `{Oid(expr)}` inserts the identifier of the object referenced by `expr`.

Note that the argument `expr` can be another pattern such as `{this}`. Specifically, the expression `{Oid({this})}` is equivalent to `{this.oid}`. However, the `Oid` pattern is also applicable to patterns not including `{this}`.

Examples are available from the CB-Forum, see http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2502545.

The above set of patterns can be extended by user-defined functions. In principle, any routine that can be called from the ConceptBase server, can also be called in an answer format. The programming interface is not documented here since this requires extensive knowledge of the ConceptBase server source code. For the experienced user, we provide an example in the subdirectory `Examples/AnswerFormat` of the ConceptBase installation directory, see files `externalcall.sml` and `externalcall.swi.lpi` (`externalcall.bim.lpi` for BIM-Prolog variant). Note that one has to create a persistent database, load the model `externalcall.sml` into it, then terminate the ConceptBase server, and then copy the file `externalcall.swi.lpi` or `externalcall.bim.lpi` into the database directory (see also appendix F). Thereafter, restart ConceptBase and call the query `EmpDept`.

3.2.6 Calling queries in answer formats

A query call within an answer format is an example of a so-called external procedure. The pattern as well as head and tail of an answer format may contain the call to a query (possibly the same for which the answer format was defined for). This allows to generate arbitrarily complex answers.

`{ASKquery(Q[subst1,subst2,...],FRAME)}`

The argument `Q` is the name of a query. The arguments `subst1,subst2` are parameter substitutions (see section 2.3). The argument 'FRAME' is the default answer format specified for the query Q. It shall be overridden in case that query Q has itself an answer format. Instead of 'FRAME' the user can also specify any answer format in the ASKquery call, even parameterized answer formats (see below) are allowed.

The effect of ASKquery in an answer format is that the above query call is evaluated and the ASKquery expression is replaced by the complete answer to the query. In terms of the substitution, the following rule is applied:

`{ASKquery(Q[subst1,subst2,...],FRAME)}` $\longrightarrow$ `X`

where `X` is the result of the query call `Q[subst1,subst2,...]` after derivation of the arguments subst1, subst2 etc. This sequencing is important since an ASKquery call can contain terms that are subject to substitution, e.g.

`{ASKquery(MyQuery[{this.cat1}/param1,{this.{x1}}/{x2}],FRAME)}`.

ConceptBase will always start to evaluate left to right and the innermost terms before evaluating the terms that contain inner terms. Hence, the derivation sequence is

```
        {ASKquery(MyQuery[{this.cat1}/param1,{this.{x1}}/{x2}],FRAME)}
⟹      {ASKquery(MyQuery[alpha/param1,{this.{x1}}/{x2}],FRAME)}          (r2)
⟹      {ASKquery(MyQuery[alpha/param1,{this.name}/{x2}],FRAME)}          (r1)
⟹      {ASKquery(MyQuery[alpha/param1,"smith"/{x2}],FRAME)}              (r3)
⟹      {ASKquery(MyQuery[alpha/param1,"smith"/param2],FRAME)}            (r4)
⟹      The answer is ...                                                 (r5)
```

where we assume the following example substitution rules

```
r1:                                {x1}   ⟶  name
r2:                          {this.cat1}  ⟶  alpha
r3:                          {this.name}  ⟶  "smith"
r4:                                {x2}   ⟶  param2
r5:  {ASKquery(MyQuery[...],FRAME)}  ⟶  The answer is ...
```

This guarantees that the query call is 'ground', i.e. does not contains terms which are subject to substitution.

The ASKquery construct allows to introduce recursive calls during the derivation of a query since there can (and should) be an answer format for Q which may contain expressions ASKquery itself. In principle, this allows infinite looping. However, the answer format evaluator prevents such loops by halting the expansion when a recursive call with same parameters has occured. The answer then contains an '~' character at the position where the loop was detected. Additionally, an error message is written on the console window of the ConceptBase server (tracemode must be at least low).

A simple example for use of ASKquery is given in `recursive-answers.sml`. The example uses a view instead of a query class in order to include also answers into the solution which not have a filler for the requested attribute, i.e. hasChild is `inherited_attribute`, not `retrieved_attribute`.

It is common practice to combine the ASKquery construct with 'Foreach' in order to display an iteration of objects in the same way. The user should define an answer format for the iterated query Q as well.

Do not mix the use of ASKquery with the view definitions in ConceptBase! The nesting depth of a view is determined by the view definition. The nesting depth of an answer generated by expansion of ASKquery is only limited by the complexity of the database. For example, one can set up an ancestor database and display all descendants of a person and recursivley their descendants in a single answer string for that person. The nested ASKquery inside an answer format usually results in the unfolding also using an answer format (possibly the same as used for the original query). This feature allows the user to specify very complex structured answers that might even contain the complete database. In particular, complex XML representations can be constructed in this way.

3.2.7 Expressions in head and tail

The features 'head' and 'tail' are similar to pattern. The difference is that any expression using 'this' (the running variable for answer objects) is disallowed. This only leaves function patterns such as ASKquery expressions and pre-defined patterns such as `{user}`. Of course, the head and tail strings can contain multiple occurences of ASKquery or other function patterns.

3.2.8 Conditional expressions

Conditional expressions allow to expand a substring based on the evaluation of a condition. The syntax is:

`{IFTHENELSE(predicate,thenstring,elsestring)}`

The 'predicate' can be one of

- `{GREATER(expr1,expr2)}`
- `{LOWER(expr1,expr2)}`
- `{EQUAL(expr1,expr2)}`
- `{AND(expr1,expr2)}`
- `{OR(expr1,expr2)}`
- `{ISFIRSTFRAME()}`
- `{ISLASTFRAME()}`

Example: `{GREATER({this.salary},10000)}`. Note that the arguments may also contain expressions. The predicate `{ISFIRSTFRAME()}` is true, when ConceptBase starts with processing answer frames. It is false, when the first frame has been processed. The predicate `{ISlASTFRAME()}` is true

when ConceptBase starts with processing the last answer frame for a given query. Otherwise, it is false. An example for conditional expressions is provided in the CB-Forum, see file "csv.sml" in http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/861803. In most cases, the IFTHENELSE construct can be avoided by a more elegent query class formulation.

3.2.9 Views and path expressions

If the answer format is defined for a complex view, then path expressions like this.cat2.cat21... for the parts of the complex answer can be defined. An example for use of answer formats for views is given in views.sml.

The reader should note that complex path expressions can only refer to components that were defined as retrieved, computed, or inherited attributes in the view definition. For example, one cannot refer to this.dept.budget in the example view EmpDept in views.sml since it is not a retrieved attribute of the dept component of the view definition EmpDept. The second expression of the answer format EmpDeptFormat uses the builtin external procedure UQ. It removes the quotes '"' from a string. Analogously, a procedure QT can be used to put quotes around a term.

3.3 Parameterized answer formats

The general way to use an answer format for a query is to define the attribute forQuery. Another possibility is to specify the answer format for a query is to use the answer representation field of the ASK method in the IPC interface.

The following code is an example for specifying a user-defined answer in the ASK method. This example is written in Java and uses the standard Java API of ConceptBase (see the Programmer's Manual for details).

```
import i5.cb.api.*;

public class CBAnswerFormat {

    public static void main(String[] argv) throws Exception {

        CBclient cb=new CBclient("localhost",4001,null,null);

        CBanswer ans=cb.ask("find_specializations[Class/class,TRUE/ded]",
                        "OBJNAMES","AFParameter[bla/somevar]","Now");
        System.out.println(ans.getResult());
        cb.cancelMe();
    }
}
```

In the example, a connection is made to a ConceptBase server on localhost listening on port 4001. The ask-method of the CBclient class sends a query to the server. The first argument is the query, the second argument is the format of the query (in this example, it is just one object name), the third argument is the answer representation, and the last argument is the rollback time.

The third argument, is the the answer representation. There are three predefined answer representations. FRAME returns the answers as Telos frames, including retrieved and computed attributes. LABEL returns only the names of the answer objects as a comma-separated list. Thirdly, default lets the CBserver choose between LABEL (for function calls) and FRAME (otherwise). Besides these pre-defined answer representations, one can specify user-defined answer formats. In our case, it is a parameterized answer format: AFParameter[somevalue/somevar]. This means that the result of the query will be formatted according to the answer format AFParameter and the variable somevar will be replaced with somevalue. The variable can be used like any other expression, i.e. it must be enclosed in {}.

The following definition of AFParameter is an example, how the parameter can be used in the pattern. If the parameter is not specified, the string {somevar} will not be replaced.

```
Individual AFParameter in AnswerFormat with
  attribute,head
     hd : "<result>"
  attribute,tail
     tl : "</result>"
  attribute,pattern
     p : "
<object>
  <type>{somevar}</type>
  <name>{this}</name>
</object>"
end
```

Note, that you can use any answer format (with or without parameters) as answer representation in the ASK method.

3.4 File type of answer formats

The optional `fileType` attribute of answer formats is used by the server-side materialization of query results (section 5.9). ConceptBase will use the specified file type when storing the query results in the file system. The default value is ”txt”. The attribute is single-valued though single-valuedness is not enforced.

Chapter 4

Active rules

Active rules specify certain actions that have to be executed if an event occurs and a condition is fulfilled at this time. Because active rules consists of an **E**vent, a **C**ondition and an **A**ction, they are abbrivated with *ECArule*.

Events (ON-part) of ECArules are insertions and deletions of objects (Tell/Untell) and queries (Ask). The events are detected during the processing of the input frames for a Tell or Untell or Ask operation. For example, if you tell 2 frames at a time, and the first frame contains an event for an ECArule, then the ECArule is executed before the second frame is processed. You can also control the sequencing of the firing of ECA rules by the so-called ECAmode and by priority orderings in the set of defined ECArules.

The condition of the ECArule (IF-part) is a logical expression over the database. It will bind free variables occurring in the condition (if any) and these bindings together with the bindings of the event are passed to the action part of the ECArule.

The action (DO-part) of an ECArule is evaluated for each evaluation of the IF-part that has is true in the database. The elements in a DO-part can be Tell, Untell, Retell, Ask and CALL actions. CALL actions can call any Prolog predicate, for example a Prolog predicate defined as a CBserver plug-in (see appendix F). Retell actions are combining an Untell and a Tell, in particular for assigning a new attribute value, e.g. `Retell((e salary newsalary))`. Optionally, one can specify an ELSE-part which consists of actions that are executed when the IF-part is not satisfiable for any binding of the free variables. The Ask action is only useful for ECArules that have an ELSE-part. In this case, the Ask will retrieve information from the database that cannot be retrieved within the IF-part. The special action 'reject' will abort the complete transaction that directly or indirectly triggered the ECArule.

The effect of ECArules is subject to the regular integrity checking of ConceptBase. If an integrity violation is detected, then the whole transaction including all updates by ECArules is rolled back. The integrity test is started after all enabled ECArules have fired.

4.1 Definition of ECArules

In ConceptBase, active rules are defined similar to query classes. The user has to create an instance of the builtin class `ECArule`. The following frame shows the Telos definition of the class `ECArule`.

```
Class ECArule with
attribute
    ecarule : ECAassertion;
    priority_after: ECArule;
    priority_before : ECArule;
    mode : ECAmode;
    active : Boolean;
    depth : Integer;
    rejectMsg : String
constraint
```

```
    { ... }
end
```

A correct ECArule must specify at least the attribute `ecarule`, the other attributes are optional. The language for `ECAassertions` is a extension of the assertion language, it is specified as text between `$` signs in the same way as rules and constraints.

4.1.1 ECAassertion

An ECAassertion has the following structure (the syntax is described in section A.3).

```
$ x1,x2/C1 y1/C2   ...
  ON event
  IF condition
  DO action1, action2
  [ELSE action3]
$
```

The ELSE-part is optional The first line contains the declaration of all variables used in the ECAassertion. The specified classes of the variables (here: `C1` and `C2`) are only used for compilation of the rule, during the evaluation of the rule it is *not* tested if the variables are instances of the specified classes. If necessary, the include predicates like `In(z,Class)` in the IF-part of the ECArule. The variables can occur in the ON-, IF-, DO-, and ELSE-part of the ECArule.

4.1.2 Events

Possible events are the insertion (Tell) or deletion (Untell) of attributes (A), instantiation links (In), or specialization links (Isa). For example, if the rule should be executed if an object is inserted as instance of `Class`, then the event statement is: `Tell(In(x,Class))`. Furthermore, an event may be a query, e.g. if you specify the event `Ask(find_instances[Class/class])` the ECA rule is executed before the evaluation of the query `find_instances` with the parameter `Class`. Potential updates to the database vaused by the ECa rules will be persistent, i.e. in such cases an `Ask` van well update the database. It is possible to use variables as a placeholders for parameters in the `Ask` event clause.

The event detection algorithm takes only extensional events into account. Events that can be deduced by a rule or a query are not detected. However, the algorithm is aware of the predefined Telos axioms, e.g. if an object is declared as an instance of a class, the object is also an instance of the super classes of this class.

4.1.3 Conditions

The condition (IF-part) of an ECArule consists of predicates combined by the logical operators 'and', 'or', and 'not'. Quantified sub-expressions (`forall`, `exists`) are not allowed. You can however use query classes to encode such sub-expressions[1]. The arguments of the predicates are either bound by the ON-part of the ECA rule or they are free variables. When an event occurs that fires an ECArule, then the condition is evaluated against the database yielding bindings for the free variables. Each such binding will be passed to the action part of the ECArule. Note that ECArules without any free variable are also possible. By default, predicates are evaluated against the old state of the object base (i.e., before the transaction started). If a predicate has to be evaluated on the new database state, i.e. the intermediate state representing the updates processed so far during the transaction, then it has to be quoted by the backward apostrophe, for example `` `(x in Class) `` instead of `(x in Class)`. The syntax `new((x in Class))` is supported as well and equivalent to the use of the backward apostrophe. Note that only conditions of ECA rules can see intermediate database states.

[1] An example for a quantified sub-expressions is `not exists y/D (x m y)`.

4.1.4 Actions

Actions are specified in a comma-separated list. The syntax is similar to that one of events, except that you can also ask queries (Ask) and call Prolog predicates (`CALL`). The standards actions are as follows:

Tell *predicate* : The predicate fact is told to the system. The predicate must be either an attribution, instantiation, or specialization predicate. All arguments of the predicate must be bound at execution time of the action.

Untell *predicate* : The predicate fact is untold from the system.

Retell *predicate* : Old facts are first untold and then the new predicate fact is told. See below for restrictions.

Ask *predicate* : The predicate is evaluated, possibly binding free variables. Note that this action can fail if the predicate is not true in the database.

CALL *proc* : The Prolog predicate matching *procedure* is called. This can bind free variables. You can define your own Prolog code as CBserver plugin.

Raise *query* : The query call *query* is raised as event. The query is not evaluated. This action can be used to trigger other ECArules that have a matching Ask event. All parameters should be bound. The *query* may not be a builtin query class or function.

noop : This action stands for "no operation", i.e. nothing is done. It can be useful for certain constructions with empty DO parts.

reject : The current transaction is aborted and rolled back.

All variables in Tell, Untell and Retell[2] actions must be bound. The insertion of an attribute `A(x,ml,y)` is only done, if there is no attribute of category `ml` with value `y` for object `x`. Then, a new attribute with a system-generated label is created. If an attribute `A(x,ml,y)` should be deleted, then all attributes of category `ml` with value `y` for object `x` are deleted[3]. It is well possible that an attribute is updated (untell+tell) several times by action parts of ECArules during a single ConceptBase transaction. Only the state of the attribute after all ECArule firings will be visible after a successful commit.

There are a few special procedures, which may be called within the DO- or ELSE-part of an ECArule:

CreateIndividual(Prefix,ID): A new individual object with the given prefix and a system generated suffix is created. The object identifier of the created object is returned in the second argument, which must be therefore a free variable. The prefix must be an existing object name (e.g. the class name) otherwise the ECArule compiler will report an error.

newLabel(Prefix,L): A new label that is not yet used as object name is created. The prefix must be an existing object name. The 2nd argument should be a variable. The type of the variable can be `Label` or a more specific class name.

CreateAttribute(AttrCat,x,y,ID): This predicate creates a new attribute for object `x` with value `y` in the given attribute category (e.g. Employee!salary). The attribute will get a system-generated label. In contrast to the action `Tell(A(x,ml,y))` the attribute is also created, if another attribute with the same attribute category already exists.

[2]Retell is currently restricted to attribution predicates `A(x,ml,y)`. It will replace the old value of attribute `ml` of `x` by `y`. ConceptBase realizes the `Retell(A(x,ml,y))` by a combination of `Untell(A(x,ml,z))` removing the first stored attribution fact `A(x,ml,z)` and a subsequent `Tell(A(x,ml,y))`. If there are several facts such as `A(x,ml,z1)` and `A(x,ml,z2)`, then only the first one is removed by the `Untell`.

[3]Note, that an object can have more than one attribute value in one attribute category, but the attributes must have different labels.

Other predicates to be invoked via `CALL` can be defined in a LPI-file (see Counter example below).

Events and actions may also be specified in a prefix syntax, e.g. `Tell (x in Class)` instead of the longer form `Tell((x in Class))`. Furthermore, there are two simple builtin actions: `noop`[4] is not doing anything (except that is succeeds), and `reject` aborts the current transaction.

If the execution of an ECArule leads to an update to the database, i.e. via `Tell` or `Untell`, then the updated database is subject to integrity checking. If a violation is detected, then the whole transaction is rolled back including the updates done by ECArules.

4.1.5 Priorities

The attributes `priority_after` and `priority_before` ensure, that this ECArule is executed *after* or *before* some other ECArules, if several rules can be fired at the same time. There can be multiple values for each of these attributes. As you may have noted one of the two attributes is redundant since `(r1 priority_after r2)` is equivalent to `r2 priority_before r1)`. Still, ConceptBase provides both. Furthermore, ConceptBase does not automatically check the consistency of the priority declarations (if r_1 is before r_2 then r_2 cannot be before r_1). ConceptBase also does not provide for the transitivity of the priority. You can however define this yourself via appropriate deductive rules.

4.1.6 Mode of an ECA rule

The mode of an ECArule determines the point in the transaction when the condition and the action of the ECArule are evaluated and executed. Possible values are:

Immediate: The condition is evaluated immediately after the event has been detected. If it evaluates to a non-empty answer, the DO-action is executed immediately, too. If the answer is empty, then the ELSE-actions are executed (provided that the ECArule has an ELSE part).

ImmediateDeferred: The condition is evaluated immediately after the event has been detected. If it evaluates to a non-empty answer, the actions of the DO-part are executed towards the end of the current transaction.

Deferred: The condition is evaluated towards the end of the current transaction. If it evaluates to a non-empty answer, the actions of the DO-part of the ECArule are executed immediately after the evaluation of the condition. Otherwise, the ELSE-actions are executed (provided that the ECArule has an ELSE part)

The answer to the evaluation of the condition of an ECArule is the set of all combinations of variable fillers that make the condition true.

The modes `Immediate` and `ImmediateDeferred` differ considerably from the mode `Deferred`: the condition of the ECArule is evaluated while not all frames of the current transaction are told (resp. untold). So, a quoted predicate like `'(x in Class)` will be evaluated against a database state in which only those frames of the current transaction are visible (resp. invisible) that were told (resp. untold) before the ECArule was triggered!

The default is `Immediate`. ConceptBase shall enforce a first-in-first-out sequencing of ECArules with modes `ImmediateDeferred` and `Deferred`. This sequence will enforce the complete execution of a triggered ECArule before the next rule triggering is handled. The strict sequencing avoids intertwining of action executions of multiple ECArule threads. So, if the answer to a condition of an ECArule has multiple entries, then the actions belonging to the respective answers are executed in a sequence in which no action of another ECArule is called.

[4] Earlier ConceptBase releases used the keyword `commit` instead of `noop`. The semantics was the same. We continue to support the use of `commit` in legacy ECArules.

4.1.7 Execution Semantics

The mode of an ECArule influences its execution semantics. There are three basic steps. First, a Tell/Untell/Ask transaction is translated in a sequence of atomic events. In case of a Tell/Untell, each frame inside the Tell induces a delimiter in the event list that is used later for the event processing. A typical event list might look like

e_1	e_2	e_3	e_4	e_5	...

Here, the events $e_1 \ldots e_3$ were generated for the first frame of a Tell/Untell, the events e_4, e_5 were generated for the second frame. The event list is right-open since the execution of actions from ECArules can lead to further events. Each event has one of the forms *Tell(lit)*, *Untell(lit)*, or *Ask(q)*, where *lit* is an attribution, instantiation, or specialization predicate, and *q* is a query call.

ConceptBase will scan the event list and process events as soon as it detects a delimiter. For example, when ConceptBase "sees" the delimiter between e_3 and e_4, it will start to process e_1 to e_3, one after the other. Processed events are removed from the event list. The first step is to determine the matching ECArules for a given event e_i. This yields a *working set* of rules for each processed event e_i:

$ws(e_i)$ = Set of all ECArules whose ON part matches e_i

The matching of the ON part of the ECArule and the event typically leads to a binding of variables in the rule's IF and DO parts. This binding is stored in the rule's representation within the work set. The rules in the working set are sorted to reflect the priority settings of ECArules. If two rules have no priority defined between them, then the definition order is used to sort them (older rules before newer rules). Each rule r in the working set will then be processed as follows, one after the other.

The rule processing depends on the mode of the ECArule in the working set. If the mode is `Immediate`, then the IF part is evaluated (yielding all possible combinations of variables that make the IF part true). If the answer set is empty, then ConceptBase will call the ELSE actions of the ECArule. Otherwise, ConceptBase will call the DO part for each variable combination determined in the previous step.

If the mode is `ImmediateDeferred`, then ConceptBase will evaluate the IF part of the current rule like before. Instead of calling the DO (or ELSE part), it will however put a term $do(r, actions)$ on a waiting queue Q. The parameter r identifies the rule. The parameter $actions$ contains all instantiations of the DO-part of r by the variable substitutions computed by the evaluation of the IF part of r. If there are no such substitutions, then `actions` is set to the ELSE part of r (if existent). Otherwise, no actions would be appended to the wating queue.

If the mode is `Deferred`, then ConceptBase will not immediately evaluate the IF part but will just append a term $def(e_i, r)$ to the waiting queue Q.

When all events in the event list are processed, i.e. when all frames in a Tell/Untell transactions are transformed and stored, or when the Ask call has been processed, then ConceptBase will start to process the waiting queue Q. It is processed in a first-in-first-out (FIFO) manner. Each `do` and `def` entry is processed according to its type. If the entry has the form `do(r,actions)`, then ConceptBase will execute the actions in the second parameter.

If the entry has the form `def(e,r)`, then ConceptBase will first determine all combinations of variables in the IF part of r. Then it will call the actions of the DO (or ELSE part) for the computed answers. Note that the event e typically binds some variables in the rule r.

Actions of an ECArule can update the database. They will then lead to new entries in the event list that are processed just like described above. The events are generated per action (Tell/Untell/Retell) and then followed by a delimiter, i.e. ConceptBase will start processing the events after each Tell/Untell/Retell action. Examples highlighting the differences of the three execution modes are presented in the CB-Forum at `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2924361`.

You can use the tracemode `high` (see chapter 6) to debug ECArules. The ConceptBase server will then write trace messages about the execution of ECArules on the console terminal.

4.1.8 Activate and Deactivate ECA rules

The attribute `active` allows the user to deactivate the rule without untelling it. Possible values are `TRUE` and `FALSE`. The default value is `TRUE`, i.e. by default are ECArules active.

4.1.9 Depth

The attribute `depth` specifies the maximum nesting depth of ECArules. ECArules may be fired by events which are produced by actions of the same or other ECArules. Because this often results in an endless loop, the execution of the ECArule is aborted if the current nesting depth is higher than the specified value. The default of this attribute is 0 (= no limitation).

4.1.10 User-definable Error Messages

If the ECArule rejects the transaction in some cases, it is useful to specify an error message. The value of the attribute `rejectMsg` is returned to the user.

4.1.11 Constraints

The constraints of the class ECArule ensure, that the attributes `mode`, `active` and `depth` have only single values and that the attribute `ecarule` has exactly one value.

4.2 Examples

The Telos source files of the following examples can also be found in your ConceptBase installation directory at `$CB_HOME/examples/ECArules`. Further examples are in the CB-Forum at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1747992.

4.2.1 Materialization of views by active rules

Materialization of views means that deduced information is stored in the object base. We provide here an example, how to materialize and maintain simple views.

```
Class Employee with
attribute
    salary : Integer
end

View EmployeeWithHighSalary isA Employee with
constraint
    c : $ exists i/Integer (this salary i) and (i > 100000) $
end

Class EmployeeWithHighSalary_Materialized
end
```

The view `EmployeeWithHighSalary` contains all employees who earn more than 100.000. The class `EmployeeWithHighSalary_Materialized` will contain the same employees. This implemented by the following ECArules:

```
ECArule EmployeeWithHighSalary_Materialized_Ins with
ecarule
    er : $ x/Employee
    ON Tell(In(x,Employee))
    IF new(In(x,EmployeeWithHighSalary))
    DO Tell(In(x,EmployeeWithHighSalary_Materialized)) $
end
ECArule EmployeeWithHighSalary_Materialized_Del with
ecarule
    er : $ x/Employee
    ON Untell(In(x,Employee))
```

```
    IF In(x,EmployeeWithHighSalary)
    DO Untell(In(x,EmployeeWithHighSalary_Materialized)) $
end
ECArule EmployeeWithHighSalary_Materialized_Ins_salary with
ecarule
    er : $ x/Employee y/Integer
    ON Tell(A(x,salary,y))
    IF new(In(x,EmployeeWithHighSalary))
    DO Tell(In(x,EmployeeWithHighSalary_Materialized)) $
end
ECArule EmployeeWithHighSalary_Materialized_Del_salary with
ecarule
    er : $ x/Employee y/Integer
    ON Untell(A(x,salary,y))
    IF In(x,EmployeeWithHighSalary)
    DO Untell(In(x,EmployeeWithHighSalary_Materialized)) $
end
```

The first rule checks, if the employee belongs to the view, when the employee was inserted. Note, that we don't use the constraint of the view in the ECArules, we just reuse the view definition here. The second rule does the same for deletion of employees.

The third rule checks, if the employee is an instance of the view class, when the attribute `salary` was inserted. Again, the fourth rule does the same for deletion of the attribute.

If the number of employees is large it is more efficient to ask for the instances of materialized than to evaluate the view. However, if updates occur quite often, materialization is not good, because materialized view must be maintained for every update transaction.

4.2.2 Counter

This example shows how to call prolog predicates with an ECArule. It implements a counter for a class `Employee`. The counter is stored as an instance of the object `EmployeeCounter`. Whenever an employee is inserted or deleted from the object base, the counter is incremented or decremented.

```
Class Employee end
EmployeeCounter end

ECArule EmployeeCounterRule with
ecarule
        er : $  x/Employee i,i1/Integer
        ON Tell(In(x,Employee))
        IF (i in EmployeeCounter)
        DO Untell(In(i,EmployeeCounter)),
           CALL(increment(i,i1)),
           Tell(In(i1,EmployeeCounter))
        ELSE Tell(In(1,EmployeeCounter))
        $
end

ECArule EmployeeCounterRule_del with
ecarule
        er : $  x/Employee i,i1/Integer
        ON Untell(In(x,Employee))
        IF (i in EmployeeCounter)
        DO Untell(In(i,EmployeeCounter)),
           CALL(decrement(i,i1)),
           Tell(In(i1,EmployeeCounter))
        $
end
```

The files `$CB_HOME/examples/ECArules/counter.*.lpi`[5] contain the code for the Prolog predicates `increment` and `decrement`. You must copy LPI files to the database directory before you start the ConceptBase server (see also appendix F). Note, that all free variables of the PROLOG predicate must be bound in its call. Furthermore, the variables must be bound to object dentifiers, if you want to use them in a Tell,Untell or Ask action.

The effect of the `increment` and `decrement` procedures can also be achieved using the arithmetic expressions like $i+1$. The simple solution with arithmetic expressions is available from `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/d2773786/EmployeeCount.sml.txt`. The purpose of the example above is only to show that user-defined PROLOG predicates can be called in the DO-part of an ECArule. You can define more interesting PROLOG predicates like sending an email to a user with content derived from the object base. This requires however some knowledge of PROLOG and of the internal features of the ConceptBase server.

You should also note that the count of a class `c` can always (and more correctly) be computed by the function `COUNT(c)`. That does even count inherited and deduced instances.

4.2.3 Timestamps

An often asked requirement in meta modeling applications is the recording of creation and modification dates. ConceptBase stores the creation time of an object in its object base, primary for the use of *Rollback queries*. With the predicate *Known(x,t)* the time of the creation of *x* can be made visible in rules or queries. The following frames shows how to use it:

```
Class Employee with
attribute
    salary : Integer;
    createdOn : TransactionTime;
    lastModified : String
rule
    createdOnRule : $ forall t/TransactionTime
         Known(this,t) ==> (this createdOn t) $
end
```

The limitation of this approach is, that it just records the creation date of an object and not the time when it was modified. i.e. the value of an attribute was changed.

To overcome this restriction, one can use an ECArule to update the attribute `lastModified` of the above example, whenever an attribute of the category `salary` is inserted.

```
ECArule LastModified_Employee_salary with
ecarule
        er : $ t1,t2/TransactionTime y/Employee i/Integer
               es/Employee!salary lab/Proposition
        ON Tell(A(y,salary,i))
        IF A(y,lastModified,t1)
        DO Untell(A(y,lastModified,t1)),
           Ask(new(In(es,Employee!salary))),
           Ask(new(P(es,y,lab,i))),
           Ask(new(Known(es,t2))),
           Tell(A(y,lastModified,t2))
        ELSE
           Ask(new(In(es,Employee!salary))),
           Ask(new(P(es,y,lab,i))),
           Ask(new(Known(es,t2))),
           Tell(A(y,lastModified,t2))
```

[5] Since ConceptBase 6.2, LPI plug-ins can be defined in two different formats. A file with the suffix .bim.lpi is intended to be used by a CBserver based on MasterProlog (formerly BIM-Prolog). If the CBserver is based on SWI-Prolog, the server reads files with the suffix .swi.lpi. Both Prolog Environments used a slightly different syntax which requires different implementations. ConceptBase 7.0 only supports SWI-Prolog, hence providing the swi variant is sufficient.

```
        $
end
```

This ECArule is executed when a new salary is assigned to an employee. The condition checks, if there exists already a value for the `lastModified` attribute. If this is the case, the attribute is deleted. Except for deletion of this attribute, the 'DO' and the 'ELSE' part of the rule are identical.

First, we have to retrieve the transaction time of the new object (es), when it was inserted into the knowledge base (t2). We have to use `new` in the ask operations, because the object was not visible before the transaction started. After that, we can tell the new attribute. Note, that the transaction time is represented as string of the form `"tt(year,month,day,hour,minute,sec,millisec)"`.

We included this example rule to show that the `Ask` action can be used in both the DO and ELSE clauses of an ECArule. A more elegant solution is however to do all querying inside the IF clause. See the CB-Forum at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/d2995330/ECA-lastmodified2.cbs.txt for an example.

4.2.4 Simulation of Petri Nets

ECArules are a powerful tool to express semantics of concepts that are not expressible by deductive rules. For example, the semantics of petri nets, in particular the firing of an enabled transition, can be expressed by a single ECArule.

```
ECArule UpdateConnectedPlaces with
  mode m: Deferred
  rejectMsg rm: "The last firing of a transition failed.
                 Check whether the transition was enabled!"
  ecarule
        er : $  fire/FireTransition tr/Transition pl/Place
                n,n1/Integer
        ON Tell (fire transition tr)
        IF (tr in EnabledTransition) and
           (pl in ConnectedPlace[tr]) and
           (pl tokenFill n) and
           (n1 = n+NetEffectOfTransition(pl,tr))
        DO Untell (pl tokenFill n),
           Tell (pl tokenFill n1)
        ELSE
           reject
        $
end
```

The example shows that the condition can also be a complex logical expression. Note that the event (ON-part) binds the two variables `fire` and `tr`. The condition (IF-part) additionally binds the free variables `pl`, `n`, and `n1`. Each binding of the free variables is passed to the DO-part leading to some update of the `tokenFill` attribute. In case that the IF-part cannot be evaluated to true, the ELSE-part is executed. That will lead to an abortion of the current transaction rolling back all updates and issuing the error message listed under `rejectMsg`. The complete example of modeling petri nets is in the CB-Forum (http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1419080). Further examples of ECA rules can be found at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1747992.

It should be noted that one can also ask queries in the DO-part (and ELSE-part) of an ECArule. Other than for the IF-part, such queries are only evaluated once.

4.3 Limitations in the current implementation

The current implementation of active rules in ConceptBase has several limitations.

- The dependency graph of ECA rules may contain cycles, i.e. rule R1 executes an action, which fires the rule R2 and R2 has an action which fires R1. The current algorithm for detecting cycles tests the dependency graph during compile time. The cycle checker prints a warning message on the console if it has detected a possible cycle, but the compilation of the rule is not aborted, because it is still possible that the cycle does not occur during the execution of the rule. The user must take care, that endless loops are avoided.
- The ON-part of an ECArule consists of a single atomic event. ConceptBase does not support complex events such as "event 1 occurs after event 2".
- Instantiations and specializations of system classes (Proposition, Attribute, InstanceOf, ...) are not detected in the event manager. Adding those events would affect the performance.
- Like QueryClasses, ECArules can not be modified after they have been told (e.g. change the event, condition or actions). The only updateable attribute of `ECArule` is `active`. If you set this attribute to `FALSE`, you can deactivate the rule for a certain time. Deleting or setting the attribute to TRUE re-activates the ECArule. If you don't want to use an ECArule anymore, you can untell it as a whole.
- The ECArule evaluator is not very efficient. Response times may be long in case that an ECArule leads to many updates to the object base.
- ECArules that update the database will temporarily disable the cache-based predicate evaluator. By this, certain recursive deductive rules called in the IF-part of ECArules may not terminate if you activate the CBserver parameter `unsafe` (see section 6). The 'unsafe' mode disables checks on the presence of recursion. This leads to faster execution time since the 'safe' mode would refresh the predicate cache after each update to the database.

Chapter 5

The Module System

The module concept provides the facility to divide a *ConceptBase* database into an arbitrary number of autonomous clusters. We use the term **module** for such a cluster. A module is autonomous in the sense that the visibility of objects and validity of deductive rules and integrity constraints is limited to the scope of a certain module. This implies that for each server operation (TELL, UNTELL, ASK) the user has to define a module in whose context he wants the operation to be executed.

One useful application of modules in ConceptBase are modelling situations where different objects are labeled with identical names. In earlier versions of *ConceptBase* this was prohibited by the *Naming Axiom* which demands that different objects have different names. Now objects with identical names can be stored in different modules without interferences. As another application users can manage alternative conceptual models of the same domain in different modules.

The set of visible objects in a module context is not limited to the set of objects defined for the module. The *ConceptBase* module concept permits to reuse existing objects from different modules via **import** and **export** interfaces. Furthermore, modules can be **nested**: a nested module is only visible within the context of its "father" module. Objects that are visible in a "father" module are visible (and can be reused) to all its "child" modules.

ConceptBase users can be assigned to so-called **home modules**, i.e. the module they start working with when registering with the ConceptBase server. So-called **auto home modules** force every user in her own module to further reduce potential unwanted interferences. A flexible **access control** mechanism allows to define access rights of users simply by query classes defined either globally or locally to a module.

5.1 Definition of modules

A `Module` is a standard O-Telos object which defines the content of a module.

```
Class Module with
 attribute
    contains: Proposition
end
```

In ConceptBase, a module is a container of objects. Modules create a name space: object names must be unique within one module, but different modules can contain different objects with identical names.

Tell, **Untell** and **Ask** operations work relatively to a specified module. The normal way of specifying the context in which a transaction takes place is using the *Set Module* function from the *CBworkbench*. See subsection 5.2 to learn how to change the module context of a transaction.

The basic set of predefined objects of *ConceptBase* (such as `Class`, `Proposition`, `QueryClass`, etc.) belongs to the predefined module `System`. The default module of clients logging into a CBserver is `oHome`, a direct submodule of `System`. You can set the module context to your other modules in order to manipulate them.

The `contains` attribute of the `Module` object is a derived attribute and is a link to all objects defined inside the context of a certain module. This attribute is not stored explicitly but can be used anywhere in the O-Telos assertion language.

Now we show how to create modules. We introduce a small running example in order to demonstrate all module-related facilities of *ConceptBase*.

Let's assume you've started ConceptBase with a fresh database file. After telling the following two frames, the server contains two new modules, which are nested inside the pre-defined `oHome` module:

```
Module Master end            Module Work end
```

5.2 Switching between module contexts

The standard way for changing the module context is to choose the *select module* menu entry from the *Options* menu in the user interface. A dialog with a listbox containing all known modules in the current context are shown. Double-clicking a module entry in the listbox sets the current module context to the selected module and lists all modules that are visible in this module (i.e., its submodules and supermodules.

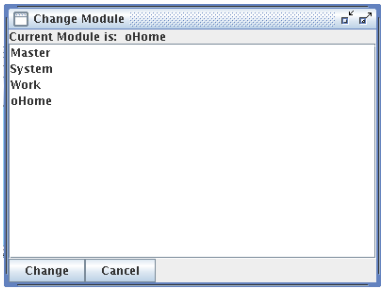

Figure 5.1: The module context selection window

In our example, you should get a window displaying the modules `Master`, `System`, `Work`, and `oHome` in alphabetic order (see Figure 5.1). Now double-click the entry `Work`. As a result the module context of the *CBserver* is set to the module `Work`. As the `Work` module has got no nested modules, there are no additional modules displayed in the listbox (i.e., all modules that are visible in the `oHome` module are also visible in the `Work` module. Alternatively, you can specify the module context using the *load model* operation and placing a

`{$set module=MODNAME}`

inline command placed before the first Telos frame of a source model file. `MODNAME` stands for the name of the module in which you wish to define the content of the source model file. Module paths like `System-oHome-Work` are allowed as well. Please note that only one inline-command is allowed within one source model file. Specifying module contexts using inline commands is a facility for automatically loading Telos frames of large applications which are spread over different modules – without requiring the user to employ the *select module* function from *CBiva*.

Let's set the module context to `Master` and then TELL the following frame:

```
Employee with
   attribute
      name : String
end
```

The object `Employee` is now only visible in the module `Master`. When you set the module context to `Work` (or `System`) and try to load the `Employee` object, you should get an error message from the server stating that the object `Employee` is not visible in that module context.

5.3 Using nested modules

A nested module is a module defined in the context of a ”father” module and therefore can only be seen in the context of the ”father”.

After the definition of the `Master` and `Work` modules as nested modules to the `oHome` module, let's define a nested module to the `Work` module. We assume that the current module context is set to the `Work` module. Now tell the following frame:

```
Module Test end
```

As a result we have defined the nesting hierarchy depicted in Figure 5.2.

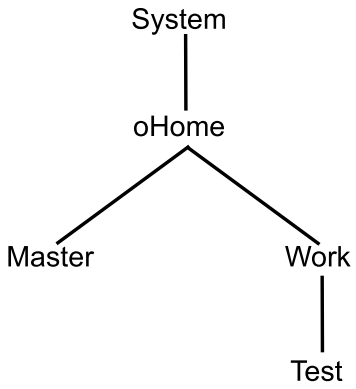

Figure 5.2: A simple nesting hierarchy of modules

A nested module can see the content of all its fathers. Therefore when you set the module context to `Test`, you can reference all objects contained in the modules `Test`, `Work`, `oHome`, and `System`. When you set the module context to `Work`, you can reference all objects contained in `Work`, `oHome`, and `System`.

5.4 Exporting and importing objects

In order to use objects which are not visible in a module, *ConceptBase* offers the export/import facility. The `Module`-object has two further attributes, namely to objects exported from a module and to modules imported by a module.

```
Module with
 attribute
    contains     : Proposition;
    exports      : Proposition;
    imports      : Module
end
```

In order to allow other modules to import objects from a module, we need to define an `export` attribute from the module object to those objects. We call the set of objects exported by a module the *export interface* of a module. In order to include the export interface of another module to a module, we need to define an `imports`-attribute from the module object to the module to be imported.

In our running example we would like to define a specialization of the Class `Employee` within the `Work` module. It is desirable to reuse `Employee` from the `Master` module instead of redefining it in the `Work` module.

In order to import the class `Employee` to the `Work` module, we have to define all objects belonging to the `Employee` class as exported objects. First change the module context to `Master`. Now TELL the following Frame:

```
Master with
   exports
      e1 : Employee;
      e2 : Employee!name
end
```

Now you have defined the objects `Employee` and `Employee!name` as exported objects of the `Master` module. Any module in your database, which defines an `imports`-attribute to the `Master` module, can now reference these objects. Now change the module context to `Work` and TELL the following Frame:

```
Work with
   imports
      i1 : Master
end
```

Now the objects mentioned above are visible in the context of the `Work` module. Check this by loading the `Employee` object with the *Load frame* function from the *CBworkbench*. You can now define the specialization `Manager` of `Employee` in the context of the `Work` module.

```
Manager isA Employee end
```

When untelling `exports`-attributes from a module, *ConceptBase* checks for integrity violation in all concerned modules. Try untelling the `exports`-attributes from the `Master` module and you should get an error message saying that referential integrity is violated in the `Work` module. The reason for this violation is natural: as the class `Employee` is no longer exported from `Master`, it is no longer visible inside the `Work` module and therefore referential integrity is violated for the `Manager` specialization of `Employee`.

5.5 Setting user home modules

When ConceptBase is used in a multi-user setting, it makes sense to automatically assign clients of ConceptBase users to a dedicated module context, their so-called *home module*. To use this feature, the database of the ConceptBase server has to contain instances of the pre-defined class `CB_User`. This class is defined as follows:

```
CB_User with
 attribute
      homeModule : Module
end
```

Assume that we have two users `mary` and `john` who need to be assigned to different modules when they log into the CBserver by their favorite user interface. The system adminstrator should then include the following definitions to the database of the CBserver:

```
Project1 in Module end
Project2 in Module end

mary in CB_User with
 homeModule m1 : Project1
end

john in CB_User with
 homeModule m1 : Project2
end
```

As a consequence, the start module of the two users will be set accordingly when they log into the CBserver. The home module feature is especially useful in a teaching environment. The teacher can put some Telos models into the shared `System` module. Students' home modules would be assigned to sub-modules, e.g. based on group membership. Each student group can then work on an assignment by working on their sub-module without interfering with other student groups.

There is a subclass `AutoHomeModule` of `Module` which addresses teaching applications of ConceptBase where by default any user should work in her own module context. Rather than having to define separate modules for each user explicitely, you can just define a certain module to be of type `AutoHomeModule`:

```
LectureModule in AutoHomeModule with
  exception e1: mary
end

mary in CB_User with
 homeModule m1 : LectureModule
end

john in CB_User with
  homeModule m1: LectureModule
end
```

Here, user `john` (a student) will be automatically be assigned to a new module `Workspace_john` that is created as sub module of `LectureModule`. User `mary`, presumably the teacher, is defined to be an exception to this rule and she will get the home module `LectureModule`. By this, one can reduce the chances of unwanted interferences between users of the module LectureModule. Still, all student members can read the definitions in the module `LectureModule`.

An even simpler way to separate the workspaces of *any* user is to define

```
oHome in AutoHomeModule with
  exception
    e1: mary
end
mary in CB_User end
```

Then, no user needs to be defined explicitely[1] as instance of `CB_User` and still will get assigned her own sub module to work in. One user like `mary` should be kept as exception because she would work by default on the `System` module to define objects that are visible to all students.

The home module definitions need to be made within module `oHome` because they will be evaluated upon client registration (server method `ENROLL_ME`) in this module context. Please note that the module context is only dependent on the user name, not on the client and not on the network location of the user. It could well be that a user `mary` is defined on multiple computers on the network and that different natural persons are identified by `mary`.

5.6 Limiting access to modules

When multiple users work on the same server, their workspaces not only need to be separated in a controlled way by means of the module feature. Users are also interested in controlling who has which rights on their workspace (=module). ConceptBase includes basic support for rights definition and enforcement via a user-definable query class `CB_Permitted`. The signature of this query class has to conform the following format:

[1] A side effect of the server method `ENROLL_ME` is that the user of the registering client will automatically be defined as an instance of class `CB_User`. The definition will be made in the context of the root module `System`.

```
GenericQueryClass CB_Permitted isA CB_User with
  parameter
    user: CB_User;
    res: Resource;
    op: CB_Operation
   ...
end
```

A `user` is allowed to perform the operation `op` on the resource `res` iff the constraint of the query is satisfied. Then, `user` is returned as answer of the query. If not, the answer is `nil` (equals empty set). Some fundamental definitions are pre-defined objects of ConceptBase:

```
Resource with end
Module isA Resource end
CB_Operation end
CB_ReadOperation isA CB_Operation end
CB_WriteOperation isA CB_Operation end
TELL in CB_WriteOperation end
ASK in CB_ReadOperation end
```

Hence, at least two operations `TELL` and `ASK` are pre-defined symbolizing write and read accesses to a resource. Modules are the prime examples of resources to be access-protected. Currently, only access to them is monitored by the CBserver.

When a user wants to switch to a new module, then she must at least have the permission to execute the operation `ASK` on it. Otherwise, the context switch is rejected. This is the main protection scheme offered to module owners. It is recommended to define the query class `CB_Permitted` in the module that needs protection.

Assume that there is a user `jonny` who wants to protect his module `Mjonny`. To do so, he would first define the module and set the module context to `Mjonny`.

```
Mjonny in Module end
```

Then, he would set the module context to `Mjonny` define his rights management policy, for example:

```
CB_Group with
  attribute
    groupMember: CB_User;
    permitted_read: Resource;
    permitted_write: Resource;
    owner_resource: Resource
end

CB_User isA CB_Group end

CB_Group in Class with
  rule
   rr1: $ forall p/Resource u/CB_User
          (u owner_resource p) ==> (u permitted_write p) $;
   rr2: $ forall p/Resource u/CB_User
          (u permitted_write p) ==> (u permitted_read p) $;
   rr3: $ forall u/CB_User (u groupMember u) $;
   rr4: $ forall u/CB_User p/Resource
               ( exists g/CB_Group (g groupMember u) and
                 (g owner_resource p) ) ==> (u owner_resource p) $;
   rr5: $ forall u/CB_User p/Resource
```

```
                ( exists g/CB_Group (g groupMember u) and
                  (g permitted_write p) ) ==> (u permitted_write p) $;
   rr6: $ forall u/CB_User p/Resource
                ( exists g/CB_Group (g groupMember u) and
                (g permitted_read p) ) ==> (u permitted_read p) $
end


GenericQueryClass CB_Permitted isA CB_User with
  parameter
    user: CB_User;
    res: Resource;
    op: CB_Operation
  constraint
    cperm: $ (
              ( not exists u/CB_User
                 (u owner_resource ~res) and
                  not (u == ~user) )
              or
              ( (~op in CB_ReadOperation) and
                (~user permitted_read ~res) )
                or
              ( (~op in CB_WriteOperation) and
                (~user permitted_write ~res) )
             )
              and UNIFIES(~user,~this) $
end
```

In the above example, access rights are granted to groups of ConceptBase users. The *owner* of a resource will always have full access via rules `rr1` and `rr2`.

Then, the user would claim ownership to the module via

```
CB_User jonny with
  owner_resource r1: Mjonny
end
```

Then, only `jonny` can switch to the module `Mjonny`. If a second user like `mary` was to be granted read permission, `jonny` would define within module `Mjonny`:

```
CB_User mary with
  permitted_read r1: Mjonny
end
```

In the above example, rights can also be granted to groups of users and then inherited to its members via rules `rr4` to `rr6`. It should be noted that the definitions of `owner_resource`, `permitted_read` and `permitted_write` are just for illustrating what is possible. ConceptBase only requires the query class `CB_Permitted` in the module where the access rights need to be enforced. If such a query class (or a local version as explained below) is not defined, then any access is permitted for any user.

When a user attempts to switch to new module context, ConceptBase checks whether the user has at least read permission, i.e. permission for executing the operation `ASK`, on the module. If permission is not granted, the user cannot switch the module context and an error message is presented.

The definition of the query `CB_Permitted` is visible in the module where it is defined and in all sub-modules of this module. One can also define a local version of the query by appending the module name to its name, e.g. `CB_PermittedMjonny`. This version is only tested for access to the module `Mjonny`. The local overrides the general version `CB_Permitted` and its function is not inherited to sub-modules.

There are plenty of ways to combine general and local versions of CB_Permitted yielding different access policies. When using access control, at least the module System should be protected. Otherwise, users could change essential definitions affecting all other users. Examples for access control policies are in the HOW-TO section of the ConceptBase Forum (http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2281940).

It is very well possible to make access to a ConceptBase database completely impossible by errors in the definition of CB_Permitted. For example, one could deny access to any operation by the following simple rule:

```
GenericQueryClass CB_Permitted isA CB_User with
  parameter
    user: CB_User;
    res: Resource;
    op: CB_Operation
  constraint
    cperm: $ FALSE $
end
```

In such cases, one has to start the CBserver with disabled access control and repair the definition if the query CB_Permitted. Access control is by default disabled (see option -s in section 6). Hence, you need to *explicitely enable* it when you start the CBserver.

Caution: The access control feature of ConceptBase avoids some *unwanted* interferences in a setting where multiple users work on the same server. The system in *not* save against malicious attacks! Neither does it prevent all unwanted interferences.

5.7 Listing the module content

The contains attribute allows to check which objects belong to a module. It can be used by a simple query that lists the module content of all modules that are currently visible:

```
ShowModule in QueryClass isA Module with
  computed_attribute
    cont : Proposition
  constraint
    ccont : $ (~this contains ~cont) $
end
```

A more sophisticated method is to use the builtin query listModule. A call of listModule without parameters will list the current module as Telos frames. You can also provide the module to be listed as a parameter, e.g.

```
listModule[System/module]
```

will list the content of the module System. ConceptBase will check read permission before a module content is listed. You can also use a module path as parameter, e.g.

```
listModule[System-oHome-Work/module]
```

A module path is formed much like a directory path in a file system. The root module is System and modules names are separated by the character '-'. The implementation of the listModule query preserves the order in whoch objects have been created. Note that the System module is defining the essential objects that ConceptBase requires to run correctly. You can list the System module but you should not change it.

5.8 Saving and loading module sources

The ConceptBase server (see section 6) has two command line options `-save` and `-load` to synchronize with the content of database modules as Telos sources files in the file system of the CBserver. The purpose of this feature is to allow an easy way to save/reload the complete content of a database using a readable source format. These sources can be modified with a regular text editor.

The source files generated by the save function include a `set module` directive that instructs the CBserver to load the file to the original module path when the load function is invoked (or when the file is loaded manually via the `load model` function of CBiva (see section). The file names of the saved/loaded module sources consist of a module path starting with the root module name `System` followed by the file type `sml`. For example, a filename `System-oHome-AB.sml` will hold the contents of the module `AB` that is a submodule to `oHome` that is a submodule of `System`.

The save function is activated when the option `-save` *savedir* is specified. The parameter `savedir` must be the path to an existing directory. If activated, the save function is invoked if the CBserver is shut down. The CBserver will save the complete module tree starting from the root module `System`. The save function is also invoked when a client tool (for example CBiva) disconnects from the CBserver. In this case, only the module tree starting from the home module of the user of that client tool is saved. In both cases, the save function is executed with the rights of the user who started the CBserver. The save function requires at least read permission for the module to be saved.

The load function gets activated when the option `-load` *loaddir* is specified. The directory *loaddir* should contain files with file names being formed just like with the save function. It load the files in alphabetic order, which makes sure that the sources of super modules are loaded before the sources of their sub modules. The import is executed once at CBserver startup with the rights of the user who started the CBserver. If a file contains an error, the loading of this module source fails. Error messages shall be displayed in the trace log of the CBserver. Note that the CBserver can be started with a non-empty database. The import of source files will be added to the already existing content of the database.

Examples:

```
CBserver -d DB1 -save /home/meee/DB1SRC
```

This command starts up a CBserver that will eventually save the module sources in the specified directory. Note that the saving takes place either at CBserver shutdown or when a client tool disconnects (partial save).

```
CBserver -u nonpersistent -save /home/meee/SRC
```

This variant will start a CBserver with a persistent database but the contents will nevertheless stored as Telos sources files.

```
CBserver -u nonpersistent -load SRC1 -save SRC2
```

This command starts a fresh CBserver (i.e. only system objects are defined) and then loads module sources from the directory `SRC1`. Then, client tools may modify the contents of the database. Finally, the module contents are saved in directory `SRC2`.

```
CBserver -d DB1 -load /home/meee/DB1SRC -save /home/meee/DB1SRC
```

The load and save directories may also be the same. Note that the save function will eventually overwrite the files that have been loaded at CBserver startup.

```
CBserver -u nonpersistent -load DB1SRC
```

This command will start a non-persistent CBserver and loads the module sources of directory DB1SRC. Depending on your installation you need to replace the command name `CBserver` by either `startCBserver.bat` or `startCBserver.sh`

Note that module sources do not contains the historic states of objects that are maintained with rollback times in the CBserver database. Hence, a persistent database is containing more information than the saved module tree and is also faster to start up compared to loading module sources. Still, the load/save function offers a simple way to keep a textual representation synchronized with the evolving database state, or to back-up/re-load a database state.

The CBserver parameter `-db` combines the function of `-d`, `-import -export`, and `-views`. Hence, all files will be accessed/updated in the database directory.

5.9 Server-side materialization of query results

Similar to the saving of module sources, the CBserver parameter `-views` enables the materialization of certain query results in the file system of the CBserver. To do so, one has to specify the queries to be materialized. Only queries with a single parameter or with no parameter can be materialized. The queries need to be listed with the module that contains the objects that match the query. Example:

```
MyModule with
  saveView
    v1: Q1;
    v2: Q2
end
```

You can also use deductive rules deriving the values for the `saveView` attribute.

The queries `Q1` and `Q2` need to be visible in the module `MyModule`. The queries need to have an answer format (section 3) defined for them (attribute `forQuery`). Assume that the query `Q1` has the single parameter `param: C1`. ConceptBase will then call the query `Q1[x/param]` for each instance `x` of class `C1`. The result is stored in a file with name `x` in the directory specified with the `-views` parameter. The file type of the file is taken from the optional `fileType` attribute of the answer format of `Q1`. The default file type is ".txt". The result of queries with no parameter is stored in files carrying the name of the query.

The materialization of query results is executed at the same events when the saving of module sources takes place, i.e. at CBserver shutdown, or when a client tool logs off the CBserver. To enable the materialization, you need to specify the target directory with the `-views` option:

```
CBserver -d MYDB -views /home/meee/MyViews
```

You can also use the CBshell utility (section 7) to extract the query results and materialize them on the *client side*. This method is more flexible but you need to program the CBshell scripts for to extract all required views. For example, the CBshell script

```
enrollMe alpha 4001
ask "Q1[x/param]" OBJNAMES FRAME Now
showAnswer
exit
```

will enroll to the CBserver running on a host named `alpha` with port number 4001 and will extract just the answer to `Q1[x/param]`. If there are more answers to be extracted, one has to employ separate scripts for each of them and execute them one after the other to save the results in separate files. The *server-side* method using the `-views` option will determine all possible fillers `x` for the parameter `param` and automatically save the results of `Q1[x/param]` in a separate file with filename `x`.

Further examples are available from the CB-Forum at http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/3097259.

Chapter 6

The ConceptBase.cc Server

The ConceptBase.cc server offers TCP/IP access to the functionality of the O-Telos languages. Specifically, it allows to TELL or UNTELL O-Telos objects and to ASK O-Telos object bases. The operations are called by clients (for example, the user interfaces described in section 8). An arbitrary number of clients can connect to a single ConceptBase server.

The ConceptBase.cc server is started[1] by a command line

```
<CB_HOME>/startCBserver.sh <params>
```

or on Microsoft Windows

```
<CB_HOME>\startCBserver.bat <params>
```

where `<CB_HOME>` is the installation directory of ConceptBase.cc.

6.1 CBserver parameters

The following parameters are available for the startCBserver (CBserver) command:

-d dbdir Set database `dbdir` to be loaded. If the database does not exist it is created and initialized with the O-Telos pre-defined objects. A database is maintained as a directory. The parameter is mandatory except when the update mode is set to `nonpersistent` (see below). You cannot start two concurrent servers which use the same database directory. To avoid this case, a file `OB.lock` is created in the database directory when the first server is started. If the server crashes during its execution, the file `OB.lock` will still exist in the directory. Before you restart the server, you might have to remove this file manually.

-db dbir Like `-d` but also sets the load/save/views directories to `dbdir`, i.e. the CBserver will automatically maintain the module sources in `dbdir` and also materialize the selected queries in the same directory. See section 5.9 for details.

-u updatemode controls update persistency. The allowed values are `persistent` and `nonpersistent`. If no database is provided by parameter "-d", then the default update mode is set to nonpersistent. Otherwise, the default is persistent. In nonpersistent mode, all updates are lost after the ConceptBase server is stopped. In persistent mode, updates are stored in the files of the database and will be available for future sessions.

[1] There are two other ways to start the ConceptBase.cc server (CBserver). One is for starting the CBserver from a mobile medium without prior installation on the host computer and the second one is to start the CBserver from within the ConceptBase.cc user interface. Details are in the installation guide distributed with ConceptBase.cc.

-U untellmode controls the way how UNTELL is executed by the server. The allowed values are `verbatim` and `cleanup` (default). In verbatim mode, the UNTELL operation will only untell the facts directly described by the O-Telos frame being submitted as argument. In cleanup mode, UNTELL will also try to remove the instantiation to the O-Telos system classes `Individual`, `Attribute`, `InstanceOf` and `IsA`. By doing so, UNTELL behaves inverse to the TELL operation. More details are explained in subsection 6.4.

-p portnr sets the TCP/IP socket portnumber for client connections to the CBserver. The value `portnr` must be between 2000 and 65535. If there is already a process using the portnumber, the CBserver will abort[2]. The default value for the portnumber is 4001.

-port portnr is the same as "-p portnr". Useful to avoid a harmless warning by the underlying SWI-Prolog system, which claims the parameter tag "-p" for itself.

-t tracemode sets the tracemode of the CBserver. It is one of `no`, `minimal`, `low`, `high`, `veryhigh`. The tracemode determines the amount of text displayed by the server during its execution. The tracemode does not influence the function but is used for debugging. The default tracemode is set to `low` (display CBserver interface calls plus their answer). The tracemode `minimal` will configure the CBserver to only trace the CBserver interface calls (without answers), and the tracemode `no` virtually disables tracing. The tracemode `high` and `veryhigh` are only useful for debugging the system. In these two modes, an unlikely fatal signal like division by zero will not directly abort the CBserver process but start a debug dialog. Enter "h" for options to diagnose the problem in collaboration with the ConceptBase developers.

-c cachemode turns on the query cache to allow recursive query evaluation. The value `cachemode` is one of `off`, `transient`, and `keep` (default). In transient mode the cache is emptied before each transaction. In keep mode, the cache is emptied when the maximum number of entries in the cache is exceeded or an update has invalidated the cache. Further details are explained in section 6.2.

-o optmode controls the optimizer for rules, constraints and queries. The value `optmode` is one of `0` (no optimization), `1` (structural optimization by exploiting builtin O-Telos axioms), `2` (optimizing join order) or `3` (combines 1 and 2). Default and recommended is 3.

-r secs automatically restarts the server after a crash. The value `secs` specifies how many seconds to wait before restart. Crashes are infrequent with the CBserver. Use this option only when you want maximum accessibility of the server. This option is not available on Microsoft Windows.

-s securitylevel enables or disables the access control mechanism of ConceptBase. The value `0` means that no access control is employed. Any user can ask, tell, untell, retell any object in any module. The value `1` enables access control. First, untelling of objects must happen in the module where the object has been defined. Second, any transaction submitted by a user to the CBserver is checked against the permission rules as defined in section 5. By default, access control is disabled. Enable it when you use ConceptBase in a multi-user setting and you want to avoid errorneous interferences between different users.

-e maxerrors sets the maximum number of errors to be displayed to a ConceptBase.cc client within one transaction. The value -1 means that no restriction is applied. Set to 0 to surpress any errors message and to a positive number to limit the displayed errors messages to that number. A low positive number can speed up the communication between ConceptBase client and server if a lot of error messages are generated. The default is 20.

-cc ccmode (predicate typing) controls inhowfar the CBserver applies strict typing of attribution predicates `(x m y)` occurring in the membership constraints of query classes. If the mode is set to `strict`

[2]On Microsoft Windows, the server will not abort if the port is already used by another server, possibly another CBserver process. The new server will take over the port and control it until it is stopped. Then, the old server which has used the port before has again control over this port.

(=default), attribution predicates without a unique concerned class[3] shall not be accepted. If the mode is set to `extended`, the search for concerned classes shall include subclasses (see section 2.2.7). If the mode is set to `off`, ConceptBase.cc also accepts queries with unstrictly-typed attribution predicates. The strict mode is preferable since it avoids certain semantic errors. Deduction rules and integrity constraints may never violate the predicate typing condition, even if the mode is set to `off`. An example for a query using non-strict predicate typing is available from the CB-Forum, see `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1270138`.

-mu mumode (multi-user mode) specifies whether the CBerver should run in multi-user mode (value `enabled`) or in single-user model (value `disabled`). By default, the multi-user mode is enabled, allowing multiple users with different user names to connect to the CBserver. In single-user mode, only clients started by the same user (identified by her name) can connect to the CBserver. The single-user mode is recommended if you want to block other users from logging into your CBserver. Since the test is done only on the user name, a malicious attacker could use your user name for an account on another computer and then successfully log into your CBserver. Use Internet firewalls to protect against such attacks.

-v vmode controls whether view maintenance rules are generated (vmode=`on`) or not (vmode=`off`). View maintenance rules are used to keep a ConceptBase.cc view up-to-date upon changes to the object base. Default value for vmode is `off`.

-mc maxcost this parameter defines the maximum cost level for a predicate in a binding path that is used to compile a meta formula (see section 2.2.9). The evaulation of a binding path yields fillers for the meta variables. Set maxcost to 10 if such a predicate should have about one free variable. Set it to 100 if if may have two free variables. Default is 100. The higher the number, the more candidate paths are generated, increasing the likelihood that a binding path is found. On the downside, a high value increases the compile time of meta formulas.

-pl pathlen set a maximum length for binding path candidates. In principle, the number of candidates can explode with the path length. Like the previous parameter, the path length influences the ability of ConceptBase.cc to compile meta formulas. The default value is 5. If you set the value to 0, then no meta formula can be compiled.

-eca emode controls the ECA sub-system. Possible values are `unsafe` (ECArules are evaluated without safeguarding recursive deductive rules), `off` (ECArules are not evaluated, even if some are defined), and `safe` (ECA rules are evaluated with safeguarding recursive rules; this is the default). Use the mode `unsafe` if none of your ECArules calls recursive predicates on the newest database state. This may lead to some speed-up.

-load dir specifies the directory from which the CBserver will load module sources at start-up time. The module sources must have file names starting with `System` and file type `sml`. Typically, they are generated via the `-save` flag in the preceding session of the CBserver. The default is `none`, i.e. no module sources are loaded at CBserver start-up.

-save dir specifies the directory into which to save certain textual excerpts of the database, in particular module listings. The parameter has the default value `none`, which disables the saving function. Currently, the CBserver only saves module listings. Each module is stored in one file with file type `sml`. The directory `dir` must exist. The module listing is performed when the CBserver is shut down (complete module tree is listed), or when a client tool logs out (home directory tree of the client tool is listed). The module listings uses the `set module` directive to enable the import of the file to the right module location.

-views dir specifies the directory into which the results of certain queries are materialized. See section 5.9 for details.

[3] The concerned class is a consequence of the predicate typing condition of section 2.2. You can roughly compare it to typing of variables in programming languages.

-rl rlmode controls the way how labels for generated formulas are shaped. The default value is `on`, instructing ConceptBase.cc to find a readable label for the generated formula. It typicall consists of the labels of the participating metaclass attributes occuring in the metaformula. If set to `off`, ConceptBase.cc will just take a system-generated label that contains a unique identifier. This is slightly less readable (if you want to inspect the generated formulas) but safe against certain possibilities of assigning the same label twice.

The trace of the CBserver can be saved by redirecting its output, e.g.

```
<CB_HOME>/startCBserver.sh -r 10 -p 4444 -t high -d MYDB >> mylogfile.log
```

The CBserver can also be started directly from the ConceptBase.cc User Interface (see section 8) and most parameters can be specified interactively. The command line version is recommended when one CBserver serves multiple users or when user interface and server shall run on different machines.

Note that some features like persistency can be disabled depending on your licence. A corresponding message is displayed in the window where you start the ConceptBase server.

A frequent error message during the startup of the CBserver is the following:

```
### FATAL ERROR:
Application is locked by hostname, PID 1234
### CBserver aborted
```

This messsage is printed if there is still a file with the name OB.lock in the database directory (option -d). The OB.lock file should avoid that two servers are using the same database directory. The file may be left over of a previous CBserver if the server was not stopped correctly (e.g. aborted by Ctrl-C or it crashed). If you get this error message, make sure that there is no other server running that uses this directory and then delete the file OB.lock. Then, the CBserver should start correctly.

6.2 The cache subsystem

Since version V5.2.4 ConceptBase.cc features a new query evaluation method, which uses a so-called fact cache to store intermediate results of predicates that are called during the query evaluation. Assume, for example, that an employee 'bill' has two projects 'p1' and 'p2'. Then, the result of a predicate '(bill hasProject x)' with variable x would be the set {(bill hasProject p1),(bill hasProject p2)} consisting of facts. We call this fact set also the *extension* of the predicate.

After a completed predicate evaluation, the cache of the predicate holds its extension. The cache speeds up query evaluation and prevents infinite loops when ConceptBase.cc evaluates recursive queries and deductive rules. Essentially, the cache-based evaluation allows to compute dynamically stratified semantics of the Datalog database underlying ConceptBase.cc. Plenty of examples for recursive rules and queries are provided in the online ConceptBase Forum.

The CBserver provides three cache modes to control the behavior during query evaluation:

-c off In this mode, the cache is completely disabled. Use this mode when your models do not include recursive rules. The mode is only provided for backward compatibility and has no advantages.

-c keep The cache is only emptied when necessary, i.e. when the cache has been invalidated by an update to the database, or when the maximum number of facts in the cache is exceeded. The maximum number is currently set to 50000. If necessary, the cache emptying takes place before a transaction. The keep mode is on average consuming more main memory than the transient mode but speeds up response time enormously in case of re-use of query results.

-c transient The cache is emptied before each transaction (ask, tell, untell, retell). A subsequent query is always evaluated starting with an empty cache. This mode is somewhat 'safer' than the 'keep' mode since it starts each query with an empty cache state. While the answer to a query is in principal independent from the cache mode, the cache mode has a certain influence on the persistence of

objects created within a transaction. Specifically, results of arithmetic expressions computed during one transaction shall be removed after the transaction when the cache mode is 'transient'. In cache mode 'keep', these objects continue to exist and are visible to future transactions.

By default, the cache mode is set to `keep`. Some statistics on cache usage are written to the terminal window of the ConceptBase server when the tracemode has been set to `high` or `veryhigh`.

Acknowledgements: Some techniques for the cache-based query evaluator are inspired by the 'tabled evaluation' [CW96]. Wo do however not delay the evaluation of negated predicates but rather re-order them at compile time to guarantee that all variables are bound at call time. Tabled evaluation is for example implemented in the XSB System [http://xsb.sourceforge.net/].

6.3 Database persistency

The default update mode is 'persistent'. In persistent mode, all changes to the database are written to the file system at the directory specified in the parameter '-d'. Persistent mode is suitable when a CBserver runs for a longer period of time and is directly updated by application programs. Typically, the class and meta class level are then left unchanged though ConceptBase.cc has in principal no restriction on this.

If ConceptBase.cc is used for testing and modeling purposes, the update mode 'nonpersistent' is an interesting alternative. We describe two scenarios.

Scenario 1: Single-user Modeling. When a user needs to model a certain application domain with classes and meta classes, she usually works with external Telos files (aka source models, file extension '.sml'). These files can include comments like usual with program source code. The recommended mode here is '-u nonpersistent' without specifying a database. The user can load the source models into such a non-persistent server and make corrections to the source files in case of errors or design changes. Here, ConceptBase is mostly used to check and analyze the models.

Scenario 2: Assignments. Assume that a teacher wants students to exercise a certain modelling task using ConceptBase.cc. Then, she would prepare some Telos files with necessary definitions (e.g. some meta classes) and load them into a persistent ConceptBase server. After that, she can restart the ConceptBase server in non-persistent mode on the same database created before. Student can then work on their extensions while the state of the database can easily be set back to the original state defined by the teacher. The module system of ConceptBase.cc can be used to support multiple students to work on the same server without interfering with each other.

The second scenario might also be useful in modeling. If there are some parts that are regarded as stable, the modeller can decide to make them persistent and only add/modify those Telos models that are still subject to change. In particular for large Telos models, this strategy saves time.

If ConceptBase.cc is used in a multi-user setting, then one can combine the update mode with the module feature (see section 5). In this scenario, multiple users access the same CBserver. A common super module (e.g. the root module `System`) carries the common objects of the users. Each user can be assigned to her own hown module (a sub module of the common super module) and create and update objects in this workspace without interfering with other users. If several groups of users shall share their definitions, then they would be assigned to the same home module. The home module may have sub modules for testing and releasing definitions. By employing access rights to modules, one can also design which user has which read/write permissions. The builtin query `listModule` allows to save the contents of a module to a Telos source file (see section 5.7).

6.4 The UNTELL operation

ConceptBase.cc realizes the concept of a historical database. The TELL operation submits O-Telos frames to the CBserver. The CBserver extracts the 'novelty' of the submitted frames and translates it into a set of P-facts to be stored in the object store. Any P-fact has a so-called *belief time* associated to it (see section 2.1). The belief time is an interval (t_1, t_2) whose left boundary t_1 is the time point when the P-fact was inserted to the object store, i.e. the time when the transaction was executed that led to the insertion of the P-fact. The right boundary t_2 specifies the time point after which the CBserver assumes the P-fact to be

not true anymore. It is initialized with 'Now' when the P-fact is created. This symbolic value is interpreted as the current time. You may also interpret such a time interval to be right open.

The UNTELL operation terminates the belief time of P-facts specified in an O-Telos frame. The value 'Now' is replaced by the time[4] when the UNTELL operation is executed.

From the user's perspective, a TELL operation is about creating some objects and the UNTELL operation is about deleting them[5]. Many users expect the UNTELL operation to be symmetric to the TELL operation, i.e. untelling a frame that has been told before should remove the frame completely. This is however not the case for the following reasons:

- An O-Telos frame being argument of a TELL or UNTELL operation is not necessarily all the information about an object but just some.
- Other objects might refer to an object told previously. An UNTELL operation would then be rejected to preserve referential integrity.
- ConceptBase.cc adds instantiation to the builtin classes `Individual`, `Attribute`, `InstanceOf`, and `IsA` depending on the type of the object.

The last reason is most significant in preventing symmetry. As an example consider the O-Telos frame (referred to as frame 1).

```
bill in Employee with
  name
   bname: "William"
end
```

ConceptBase.cc will recognize that `bill` is an individual and that `bill!bname` is an attribute. This information is attached internally to the P-facts, more precisely, it is derivable from the structure of the corresponding P-facts. When you ASK the system for the frame of `bill` after the TELL operation on frame 1, you will get frame 2:

```
Individual bill in Employee with
  attribute,name
   bname: "William"
end
```

Hence, the instatiation to the four builtin classes is added to the frame. If we submit the original frame 1 to an UNTELL operation, ConceptBase.cc assumes by default that only two facts should be untold:

1. The instantiation of `bill` to `Employee`.
2. The instantiation of the attribute `bill!bname` to its attribute category `Employee!name`.

As a consequence, the object `bill` and its attribute continue to exist after the UNTELL on frame 1. It would look like (frame 3):

```
Individual bill with
  attribute
   bname: "William"
end
```

[4] ConceptBase takes the system time using timezone Coordinated Universal Time (UTC) of the computer on which it is running and rounds it to milliseconds. The time is captured when the transaction is initiated, i.e. all P-facts told or untold in the transaction will use that transaction time.

[5] 'Deleted' objects can however be recovered by setting the so-called roll-back time before an ASK transaction is issued. Only ASK transaction are allowed on historical database state. It makes little sense to update a historical state. That would be 'falsifying' the history of stored P-facts.

Only by untelling this frame 3 as a second operation or by untelling the completed frame 2, the object `bill` and its attribute `bill!bname` shall be made historical.

This assymmetry of TELL and UNTELL is regarded by some ConceptBase.cc users as unnatural behavior. They expect that an UNTELL is also supposed to affect the objects themselves, not just their instantiation to classes. To support those users, we provide a so-called untell mode (see parameter -U in the list of CBserver command line parameters). The untell mode is `verbatim` will cause ConceptBase.cc to behave as explained above. The mode `cleanup` (default) will take care to remove the objects themselves provided that they have no other instantiations to classes except instantiation to the four builtin classes `Individual`, `Attribute`, `InstanceOf`, and `IsA`. Furthermore, no other object may be linked to the object subject to be untold.

6.5 Memory consumption and performance

ConceptBase.cc stores objects in a dedicated object store maintained in main memory. A P-fact $P(o, x, n, y)$ consumes about 800 bytes of main memory. That means that one can store roughly 1 million P-facts in 1 GB of main memory. A typical Telos frame is stored with roughly 10 P-facts. Hence, 1 GB of main memory allows you to store around 100.000 Telos frames. On 32 bit CPUs, this results in a maximum of roughly 400.000 frames that can fit into 4 GB of addressible main memory. This restriction virtually vanishes with 64 bit CPUs.

A single TELL/UNTELL operation submitted to the CBserver should not contain more than about 2000 frames (at about 5 attributes per frame). Otherwise, the compiler can run out of stack memory.

The raw performance of the object store, i.e. the time needed to reconstruct a frame for a given object identifier, is virtually independent from the number of P-facts that it stores. However, if you have defined many rules or integrity constraints, the performance may well degrade significantly with the number of stored P-facts. The same holds for queries. We tested the response times for standard queries such as computing the transitive closure in relation to varying database sizes. Results indicate that ConceptBase apparently approximates in many cases the theoretic optimum.

The performance of the active rule evaluator (section 4) is currently rather limited. We measured around 100 rule firings per second. This could be a bottleneck when your active rules can trigger each other.

Chapter 7

The CBshell utility

The ConceptBase.cc Shell (CBshell) is a command line client for ConceptBase.cc. It allows to interact with a CBserver via a text-based command shell. Moreover, it can process commands from a script file without further user interaction. The utility can be employed to automate certain activities such as batch-loading a large number of Telos models into a CBserver, or to extract certain answers from a CBserver.

7.1 Syntax

```
<CB_HOME>/bin/CBshell [-l] [-f scriptFile] [-t] [-i]
<CB_HOME>\bin\CBshell.bat [-l] [-f scriptFile] [-t] [-i]
```

Make sure that the variable `CB_HOME` in the CBshell command file is set to the installation directory of ConceptBase.cc. This is done automatically by the CBinstaller utility described in the Installation Guide.

7.2 Options

-l This options instructs CBshell to write errors and some statistic information to the files error.log and stat.log.

-f scriptFile Execute the commands specified in scriptFile rather that requesting commands from the command line interface. If the -f option is used and a scriptFile is specified, the commands of the file will be executed without user interaction, and CBShell will exit at the end.

-t This option can only be used in combination with the -f option. It shall instruct CBshell to confirm each comman in the script file before it is executed.

-i This options starts is for invoking a CBserver compiled directly from its sources. For developers only.

7.3 Commands

startServer *serveroptions* starts a CBserver with the specified options and connects to it

enrollMe *host port* connects to an already running CBserver, i.e. not using the startServer command

cancelMe disconnects from a CBserver

stopServer stops the CBserver which is currently connected

tell *frames* tells frames to the CBserver

untell *frames* untells frames from the CBserver

retell *untellFrames tellFrames* untells and tells frames to a CBserver in one transaction

tellModel *file1 file2* **...** tells files to the CBserver; the files can have file types ".sml" and ".txt"; if no file type is specified, ConceptBase.cc will append the default file type ".sml" to the file name

ask *Query* **[***QueryFormat* **[***AnswerRep* **[***RollbackTime***]]]** asks a query; possible query formats are `OBJNAMES` and `FRAMES`; the answer representation can be `LABEL`, `FRAME`, `default`, or a user-defined answer format; the rollback tome should be `Now`. If the query format is `OBJNAMES`, then *Query* is a string containing a query call (or a comma-separated list if query calls). It the query format is `FRAMES`, then *Query* is a string of Telos frames including a query definition. The answer representations are discussed in section 3.

hypoAsk *frames Query* **[***QueryFormat* **[***AnswerRep* **[***RollbackTime***]]]** tells frames temporarily and asks a query

lpicall *lpicall* executes the LPI call; only for debugging purposes

prolog *prologstatement* executes the Prolog statement; only for debugging purposes

getErrorMessages gets error messages for the last transaction and prints them on stdout

result *completion result* compares the given result with the last result which has been received; this command hence can be used to check whether the CBserver produces the expected completion (ok, error) and result; use this command in combination with the option `-l`

setModule *module* changes the module context of this shell

showAnswer print the last result on standard output; this can be useful if you employ the CBserver as a generator in a shell pipe (see Graphviz case below)

exit exits the shell (also stops a server which has been started in this shell)

Command arguments with white space characters have to be enclosed in double quotes (""). Command arguments may span multiple lines.

If an argument contains a string of the form `$PropName`, it will be replaced with the value of the corresponding Java property (which may be defined using the -D option of the Java Virtual Machine), if the property is defined.

The CBshell utility can be used in Unix pipes to extract textual output The CBserver and pass it to subsequent programs as input. To do so, you should start the CBserver with tracemode `no` and using the `showAnswer` command of CBshell to specify the elements to be written to standard output. The CBshell script below realizes the extraction of Graphviz [`http://graphviz.org`] specifications from ConceptBase.cc:

```
startServer -u nonpersistent -t no
tellModel ERD-graphviz2
tellModel UniversityModel
ask "ShowERD[UniversityModel/erd]" OBJNAMES FRAME Now
showAnswer
exit
```

The complete example is available from the CB-Forum at `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2519759`. Another resource for CBshell scripts is the list of test scripts at `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2596438`.

Chapter 8

The ConceptBase Usage Environment

The ConceptBase User Interface consists of two main applications:

CBiva (or CBworkbench) is the ConceptBase Interface in Java that supports the editing of Telos frames, displays instances of Telos objects, etc.

CB Graph Editor (or CBEditor) is a graphical browser for Telos objects. Telos objects can be represented by differented graphical types. Insertion and deletion of Telos objects is also supported.

The interface is based entirely on Java, so it should be usable on all platforms with a Java 1.4 or 1.5 virtual machine[1]. The Java interface includes a graphical browser and editor.

Both applications, CBiva and Graph Editor, can be run as a stand-alone Java application or as an applet inside a web browser. The use of CBiva as applet requires a browser with Java Plugin 1.4 or 1.5. If you install a Java Runtime Environment on your machine, the plugin is usually also installed and configured for your browsers. Linux users might have to install the plugin manually into a dedicated directory of their web browser if they want to create web pages with CBiva or the Graph Editor.

8.1 CBiva

CBiva is similar to the old CBworkbench used in earlier releases of ConceptBase. If you have installed a Java 1.4/1.5 virtual machine, you can start the Java user interface by the command

```
<CB_HOME>/startCBiva.sh
```

on UNIX/Linux platforms, or

```
<CB_HOME>\startCBiva.bat
```

on Windows platforms, where `<CB_HOME>` is the installation directory of ConceptBase[2]. After a few seconds, the CBiva main window should pop up. If an error occurs you might you have to edit the script/batch file so that the correct Java Virtual Machine can be found. Figure 8.1 shows the main window connected to a server and gives a short description of the buttons in the tool bar. The user interface uses the MDI (multiple document interface) model, i.e. all windows are displayed as subwindows in the main window.

The main window consists of a menu bar, a tool bar with a button panel, the area for subwindows and a status bar. The first sub window, a Telos editor, can not be closed, because it also contains the history window, which records all operations for later reuse. In the following, each component of the user interface is explained.

[1] CBiva is compatible to Java 1.4 and Java 5. It is also compatible with Java 6 Update 12 or earlier.

[2] Note, that you might have to edit these files, so that the correct Java Virtual Machine is used and the environment variable `CB_HOME` is set correctly. See Installation Guide for details.

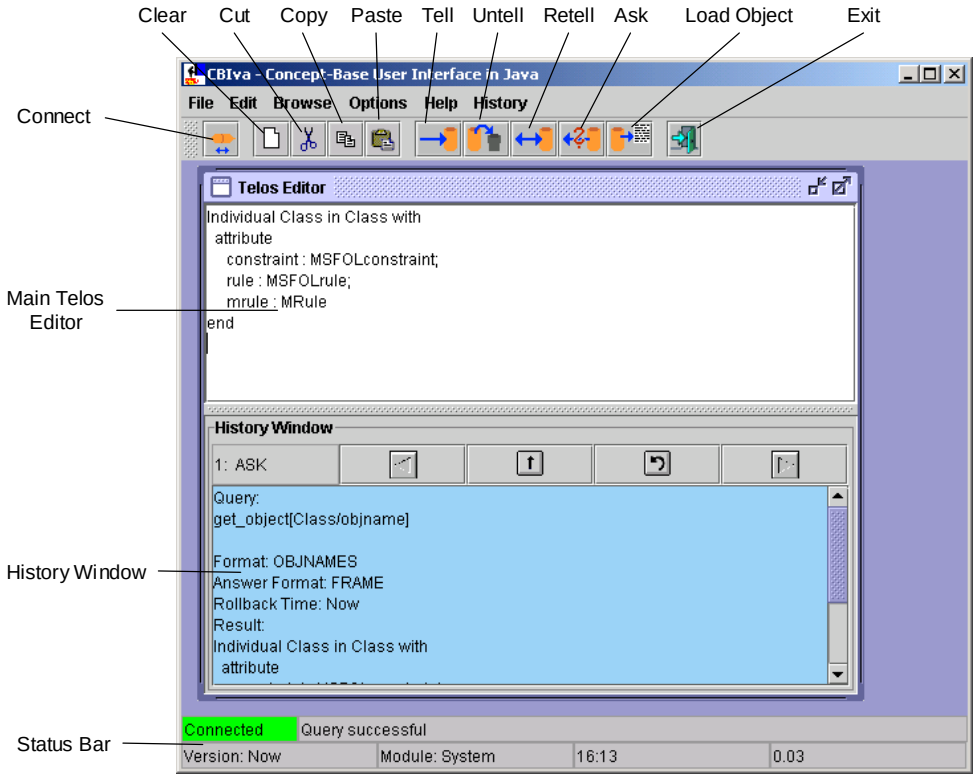

Figure 8.1: Main Window of CBiva

8.1.1 The menu bar

File: Connect: Connect to a ConceptBase server started in another shell/command window (see section 6)

Disconnect: Disconnect from a ConceptBase server

Load & Save Telos Editor: Load or Save the contents of the current Telos Editor

Load Model: Load a source model file (*.sml) into the server. As client and server are not supposed to share the same file system, this method is now implemented as a normal Tell method. The client reads the contents of the file into a string and sends the string to the server.

Start CBserver: Opens a dialog and asks for the parameters to start a ConceptBase server. If the information has been entered, the server will be started and its output will be captured in a separate window. The workbench will be automatically connected to that server[3].

Stop CBserver: Stops the ConceptBase server, only allowed for the person who started the server

Close: Close the CBiva Window, a CBEditor window remains open if available

Exit: Exit CBiva and CBEditor

Edit: Clear: Clears the text area of the currently activated Telos editor

Cut,Copy,Paste: Cut, copy or paste text to/from clipboard

Tell,Untell: (Un-)Tells the text in the currently activated Telos editor

Retell: A new window will popup and ask you to enter the frames that should be untold and told. The contents of the current Telos Editor will be inserted as default into the two text areas.

[3]This is *not* the standard way to start the ConceptBase server. Normally, the ConceptBase server is started in a separate shell/command window as explained in section 6. The standard way offers more options and control over the ConceptBase server. See also the installation guide for a discussion on the various ways to start ConceptBase.

Ask Frame: Temporarily tells the content of the Telos editor, extracts the query names from the frames, asks them as query calls without parameters, and returns the result in the Telos editor window

Ask Query Call: The query calls[4] (see section 2.3.1) listed in the Telos editor are asked and the result is returned in the Telos editor window

Load Object: Load the Telos frame of an object into the Telos editor

View Object as Tree: Opens a new window and shows the object in tree browser

Browse: New Telos Editor: Opens a new Telos editor (see section 8.1.4)

Display Instances: Opens the display instances dialog (see section 8.1.6)

Frame Browser: Opens the frame browser window (see section 8.1.7)

Display Queries: Shows all queries stored in the current database and provides a facility to call these queries (see section 8.1.8)

Query Editor: Opens the query editor (see section 8.1.9)

Graph Editor: Opens the CB Editor (graphical browser, see section 8.2). If the interface is connected to a server, the graph editor will also establish a connection to this server and ask for the graphical palette, the initial object to be shown, and the module context (see section 5). Otherwise, the graph editor will start with no connection.

Options: Set Timeout: Set the number of milliseconds the user interface waits for a response of the server.

Select Module: Select the current module (see section 5).

Select Version: Select or create a new version. A version is a special object that represents the state of an object base at a specific time. By default, all queries are evaluated on the current state of the object base (version "Now"). By selecting another version, queries are evaluated wrt. to a previous state of the object base.

Pre-Parse Telos Frames: If enabled, the user interface parses the contents of a Telos editor before it is send to the server. Thus, syntax errors might be already detected at the client side.

Use Query Result Window: If enabled, the result of a query is shown in a separate window in a table view.

Look and Feel: You can switch the look and feel to an other style:

Windows: Like Microsoft Windows

Motif: Like Motif

Metal: Default Look and Feel (Swing)

Help: ConceptBase Manual: Opens a window with the online-version of this ConceptBase manual

About: Shows a dialog with information about this program

License: Displays the license of ConceptBase in a new window

History: Load History: Load previously saved contents of the history window.

Save History: Save contents of the history window to a file.

Redo History: Redos certain operations which are currently in the history. The operations can be selected from a list.

Set History Options: Select the type of operations that should be displayed in the history window .

[4] Usually, one only asks a single query call like `Q[v1/p1]`. However, ConceptBase also supports comma-separated lists of query calls like `Q1[v1/p1],Q2[c1:p2]`. Such lists of query calls are evaluated one after the other. The results is merged into a single answer. For technical reasons, calls to builtin query classes like `get_object[Class/objname]` may only occur as a singleton.

8.1.2 The tool bar

The tool bar is the button panel below the menu bar. All buttons have tool tips, i.e. small messages that show the meaning of these buttons. The tool tips appear, if you move your mouse pointer over the button and do not move your mouse for about one second. The buttons are shortcuts for some operations that are frequently used and are also available in the menu. The operations apply to the Telos Editor which has currently the focus.

- Connect to previously connected server or disconnect
- Clear
- Cut, Copy, Paste
- Tell, Untell: The objects specified in the active Telos Editor will be added or removed from the object base.
- Retell: A new window will popup and ask you to enter the frames that should be untold and told. The contents of the current Telos Editor will be inserted as default into the two text areas.
- Ask: Evaluate the query specified in the Telos Editor. [5] If you specify the name of an ordinary class (i.e. not a query class), then ConceptBase will interpret this as a query call to find all instances of that class. You can also enter an arithmetic expression like

  ```
  100*COUNT(QueryClass)/COUNT(Class)
  ```

  .
- Load Object: Load the Telos frame of an object
- Exit

You can move the tool bar outside the main window or display it in vertical form, if you click inside the tool bar and drag it to another place.

8.1.3 The status bar

The status bar contains some fields that display general information about the status of the application.

- Connection status, either connected or disconnected from server
- Short message, usually the result of the last action
- Current Version selected, the rollback time specified for queries (default: Now)
- Current Module selected, the data module to which TELL/ASK operations are applied
- Current Time in time zone UTC (GMT), i.e. Coordinated Universal Time without summer time adjustment
- Time in seconds used for the last transaction

8.1.4 Telos editor

The Telos Editor is an editable text area, where you can edit Telos frames. The operations can be executed from the menu bar or the buttons in the tool bar. Furthermore, the text area has a popup menu on the right mouse button with the following items.

Display Instances: Displays the instances of the currently marked object

Load Object: Loads the Telos frame of the currently marked object into the Telos editor

[5] The query can either be a query call referring to an existing query or a frame representing a new or existing query definition.

View Object as Tree: Opens the tree browser with the currently marked object as root

Display in Graph Editor: Shows the object in the current window of the graph editor. The graph editor has to be started before (menu Browse)

Clear,Cut,Copy,Paste: same as in the menu bar

8.1.5 History window

The history window is part of the main Telos editor. It stores all operations and their results, so that they can later be used again. The buttons scroll the history back or forward, copy the text into the Telos editor or redo the operation in the history window (see figure 8.2). If the current operation is an "ASK", then a single click on the copy-button will copy the query to the Telos Editor, and a double click will copy the result of the query. The size of the history window can be reduced by using the slider bar between the Telos editor and the history window.

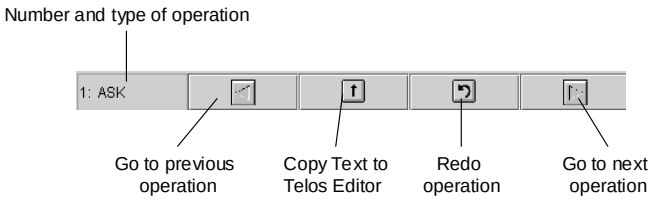

Figure 8.2: Buttons of the History Window

8.1.6 Display instances

This dialog displays the instances of a class. The class is entered in the text field. When you hit return or press the "OK" button, the instances of this class will displayed in the listbox.

If you double click on an item in the listbox, the instances of this item will be displayed. The frame of a selected item can be loaded into the Telos editor by clicking the "Telos Editor" button. A history of already displayed classes is stored in the upper right selection list box. The "Cancel" button closes the dialog.

8.1.7 Frame browser

The Frame Browser (see figure 8.3) shows all information relevant to one object in one window. The window contains several subwindows with list boxes that show super- and subclasses, the classes, the instances, attributes and objects refering this object. In the center of the window, a small window with the object itself is shown. To view the attributes of the object, you must first select the attribute category in the subwindow "Attribute Classes".

The Frame Browser can be used with and without a connection to a ConceptBase server. If it is not connected, it retrieves the information out of a local cache, which can be loaded from a file by using the "Load" button. The file has to be plain text file with Telos frames. All objects in the cache can be saved into a text file as Telos frames with the "Save" button. The contents of the cache can be viewed with the "Cache" button. The result of a query can be added to the cache by using the "Add query result" button.

The button "Telos Editor" inserts the Telos frame of the current object into the Telos Editor.

8.1.8 Display queries

This dialog displays all visible queries[6] stored in the current object base (see figure 8.4). From the list box, you can select a query and "ask" it or load its definition into the Telos editor.

[6]Visible queries are those queries that are not instantiated to the class `HiddenObject`. Certain system queries are thus excluded from the display because their purpose in an internal one.

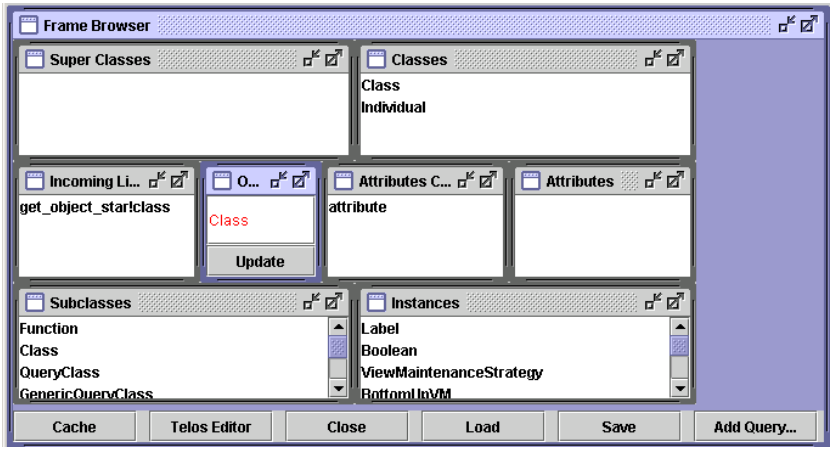

Figure 8.3: Frame Browser

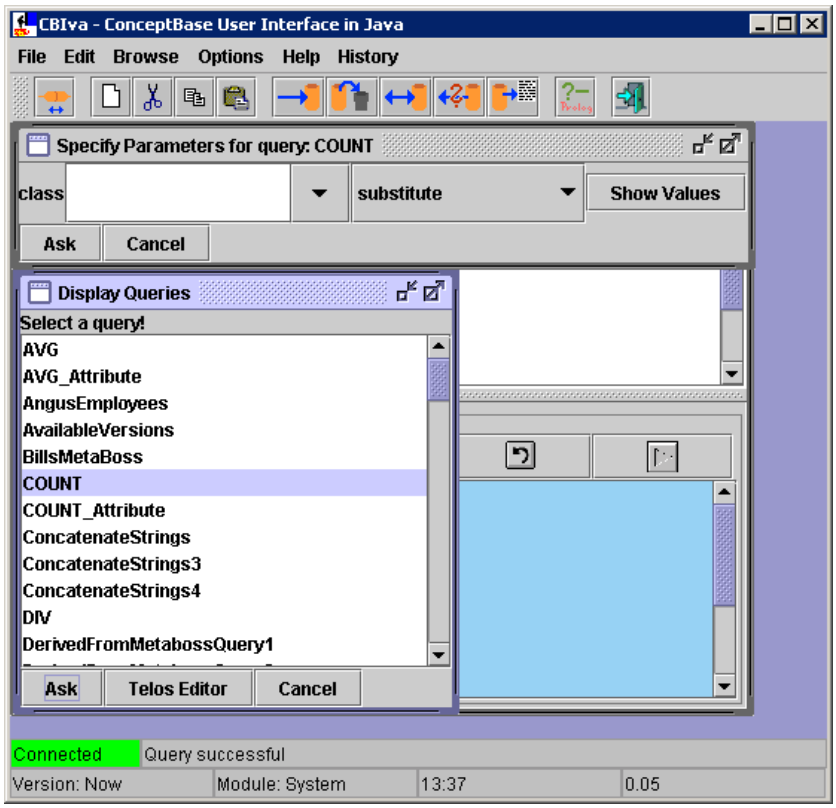

Figure 8.4: Display queries dialog

If you “ask” a generic query class with parameter, another dialog will ask you to specify the parameters. For each parameter, you can specify whether the value entered should be used as “substitute” for the parameter or as a “specialization” of the parameter class (see section 2.3). You can select a value for the parameter from the drop-down list if you have clicked on the “Show Values” button. Note that this list might be very long. Especially for the predefined queries it usually returns all objects in the database as any object can be used as a parameter for these queries.

8.1.9 Query editor

The Query Editor (see figure 8.5) allows the interactive definition of queries. The name of the query is entered in the upper left text field, the super class in the upper right field. After you have entered this information, the list box "Retrieved Attributes" will be filled with all available attributes.

Now, you can select the attributes you want to have in the result. For selection of more than one attribute, you must press the CTRL key and select the attribute with a mouse click at the same time. All attributes can be deselected by the popup menu.

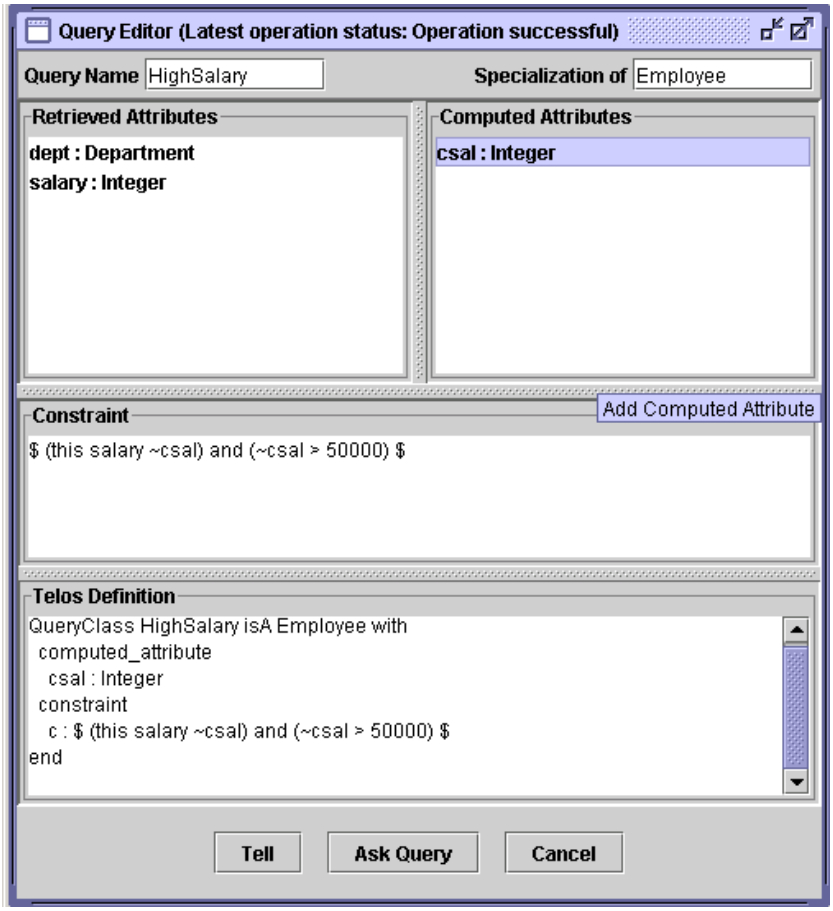

Figure 8.5: Query Editor

In the right listbox you can add computed attributes. The right mouse button brings up a popup menu, which lets you add or delete an attribute.

In the text area below the two list boxes, you can add a constraint in the usual CBQL syntax. The constraint must be enclosed in $ signs.

The text area below, shows the Telos definition of the query and is updated after every change you have made. If your query is finished, you can press the "Ask query" to test the query, i.e. it is told temporarely and the results are shown in separate window. If you are satisfied with the result you can press the Tell button to store the query in the object base.

8.1.10 Tree browser

The Tree browser (see figure 8.6) displays the super classes, classes and attributes of an object in a tree. To start the tree browser, you must mark an object in the Telos editor and select the item "View Object as Tree" from the popup menu or the Edit menu.

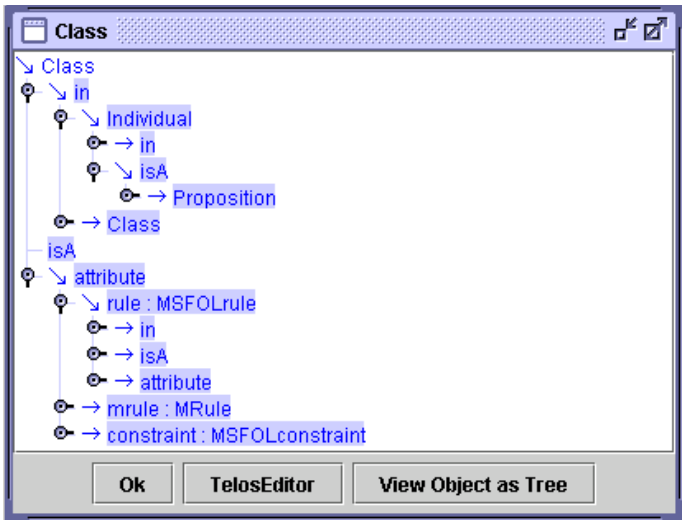

Figure 8.6: Tree Browser

To expand an item, just double click on the icon. If you mark an object name in the tree, you can load into the Telos editor with the "Telos Editor" button or open a new tree browser with the button "View Object as Tree".

8.2 ConceptBase graph editor

The ConceptBase Graph Editor is an advanced graphical modelling tool that supports the browsing and editing of Telos models. As the graphical browser of the previous releases, it supports different graphical types, i.e. different object types may be represented by different graphical objects. In addition to some predefined graphical types, the user can add his/her own graphical types by modifying and adding certain objects in the knowledge base. Furthermore, the standard components can be replaced by own classes implementing specific application-dependent behaviour.

In the following, we first give an overview of the Graph Editor application and then present the main components and functions of the Graph Editor. Details about the use of graphical types can be found in Appendix C. An example for the definition of graphical types of the Entity-Relationship model is given in Appendix D.2.

8.2.1 Overview

The ConceptBase Graph Editor is entirely written in Java. It can therefore be used on any platform with Java 1.4. The use as applet is also possible wrt. to the usual security restrictions of applets.

The Graph Editor is integrated with CBiva, i.e. Telos frames of objects shown in the Graph Editor can be loaded directly into a Telos editor and vice versa. The Graph Editor implements a Multiple Document Interface (MDI), i.e. inside the main window several internal frames can be opened. Each frame has a separate connection to a ConceptBase server. Thus, within a Graph Editor you can establish multiple connections to the same server or to different servers.

The communication with the ConceptBase server is done using standard ConceptBase queries and a special answer format. For the presentation of objects, the Graph Editor needs to know also the graphical type of an object which is included in the XML document. This document is sent by the server as answer to a query. Together with the caching techniques, the use of XML reduces the communication between client and server.

The new Graph Editor supports also the concept of "graphical types" as the graphical browser in the previous releases of ConceptBase. However, the idea of different graphical types has been extended and

makes use of some dynamic features of the Java programming language (e.g., dynamic loading and binding). Figure 8.7 gives an overview of the ConceptBase Graph Editor. Three internal windows have been opened, the two windows on the left have been connected with the same server running on "localhost", port 4001. The small window in the upper right corner is not connected to a server, but a few objects have been created.

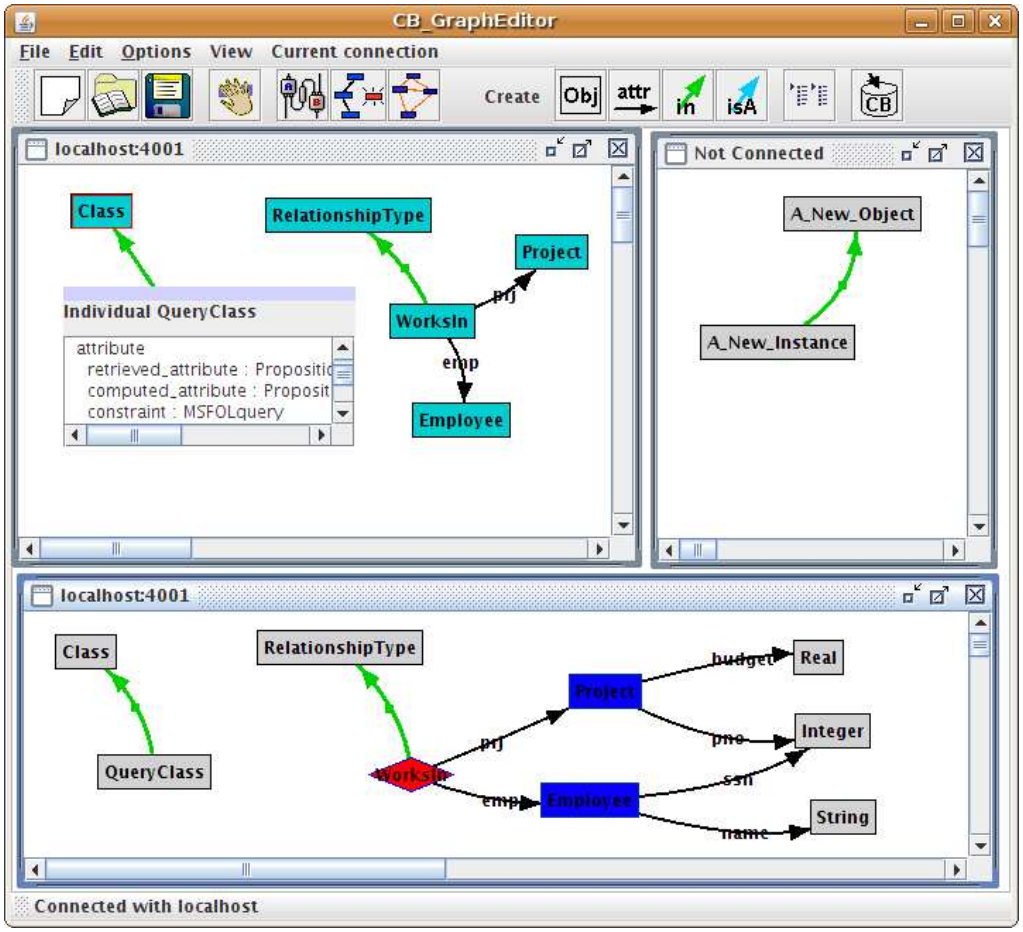

Figure 8.7: ConceptBase Graph Editor

The windows which are connected with the server, show the same model but in different representations. This is caused by the fact, that for the upper window, the default grapical palette has been chosen, and for the lower window, a customized graphical palette specifically designed for the ER model has been selected (see appendix D.2 and C).

Furthermore, you can notice that the object "QueryClass" is represented by two different components. In the upper view, the detailed *component view* has been activated by a double-click on the object. It shows the Telos frame of the object. By a double-click on the title bar of this component, one can switch back to the default view of this object. Thus, each object can be shown by a small component (the default view) and a large component which gives more detailed information. Components are in this context specific Java objects, namely instances of javax.swing.JComponent. Thus, different components can be provided to represent an object (e.g., tables, buttons, text fields). You can implement your own component and integrate into the Graph Editor by extending a specific Java class. Details about the customization of the Graph Editor using graphical types and other components can be found in appendix C.

Finally, the Graph Editor removes some drawbacks from the previous graph browser. On the one hand, it can be used to edit Telos models (see section 8.2.6). On the other hand, it is able to show also implicit relationships between objects which have been derived by rules or Telos axioms. For each type of relation-

ship (instanstation, specializations, and attributes), one can choose to see only the explicit relationships or to see all relationships.

8.2.2 Starting the graph editor

The Graph Editor is usually invoked by CBiva (the CB workbench) by using the menu item *Browse* → *Graph Editor*.

If CBiva is connected with a CB server, the Graph Editor will be connected to this server and you will be prompted to enter the object name you want to start with and a name of a graphical palette. The graphical palette is a Telos object which represents a set of graphical types which will be used to visualize Telos objects (see Appendix C). On startup, the Graph Editor retrieves all information about the graphical types from the server. Depending on your system, this might take about 10 seconds.

If CBiva has no connection with a server, the Graph Editor will be started without a connection and no internal window will be opened within the Graph Editor.

It is also possible to start the Graph Editor directly from the command line without starting CBiva. The main class of the Graph Editor is i5.cb.graph.cbeditor.CBEditor. This class has to be given as argument on the command line when you invoke the Java Virtual Machine, e.g.

```
java -classpath $CB_HOME/lib/classes/cb.jar i5.cb.graph.cbeditor.CBEditor
```

8.2.3 Menu bar

The menu bar provides access to the most important functions of the Graph Editor.

File menu

Connect to server: Connect to a new server. You can have multiple connections to one server or different servers at the same time. Each connection will be represented in one internal window with the host name and port number in the title bar of the window.

The connection dialog consists of two tabbed panes. In the first one (Address), you can enter the host address (name or IP number) and the port number. In the second pane (Initial Object), you can specify the initial object to start the browsing process and the graphical palette.

Start Workbench: Start a CB workbench (aka CBiva). If you started the Graph Editor directly or if you have already closed the workbench window, you can (re-)start the workbench by this menu item.

Save: Save the current layout of the graph into a file. The current nodes, their location and the links will be saved into a file which can be reloaded later. If the option "include graphical types" for saving layouts is enabled (see Options menu), then the graphical types will also be saved into the file. This option is useful for offline usage of the Graph Editor (i.e., without a server) as all information necessary to visualize the graph will be included in the file. By default, the file will get the extension "gel" (Graph Editor Layout).

Load: Load a layout that has been saved with the previous menu item. Existing nodes and links in the current window will be erased.

Print: Print the current graph. If the graph is larger than the page size, it will be automatically reduced to fit into the page. Thus, all printouts will be on one page.

Save image of graph: Saves an image of the current graph as PNG or JPG file. The PNG file format should be preferred as it delivers better results.

Close: Closes the Graph Editor. A CBiva window will be still available if it has not been closed before.

Exit: Exit the Graph Editor and CBiva. This operation will close both windows of the ConceptBase User Interface and exit the program.

Edit menu

The operations in this menu have an effect on the selected objects. You can select an object by clicking on it with the left mouse button. Multiple selection is possible dragging a rectangular area while holding down the left mouse button or by holding the Shift-key and clicking on objects.

Erase Selected: This option will remove the currently selected nodes and edges from the view. This operation has no effect on the database.

Selection: With this submenu, you can either select all objects, all nodes or all edges in the current frame. Furthermore, you can clear your current selection.

Options menu

The options will be stored in a configuration file (see section 8.5) when you exit the Graph Editor.

Language: The text for menu items and buttons is available in two languages (German and English). With this option you can switch between the languages. Because of unknown reasons, this option does not work under Microsoft Windows.

Saving of Layouts: This option determines whether the definition of the graphical types should be stored also in the layout file, when a layout is saved. This option is useful for offline usage of the Graph Editor (i.e., without a server) as all information necessary to visualize the graph will be included in the file.

Background Color: Here you can change the background color of the graph to your favorite color.

Component View: With this option, you can configure the view of an object if the component view is activated by a double click or by the popup menu. By default, a tree-like representation of the object with its classes, instances, super classes, subclasses, outgoing and incoming attributes is used. If you select "Frame", then the Telos frame of the object will be shown in a text area. (see figure 8.8)

Invalid Telos Objects: The Graph Editor can validate objects that are currently shown in the graph, i.e. it checks whether the object is still valid in the database or it is has already been removed (see Current Connection menu). If you select "Mark" here, then the objects will be marked with a red cross as invalid. "Remove from display instantly" will remove the objects directly from the view.

Popup Menu: These options control the behaviour of the Popup Menu (see section 8.2.5). The delay is the time (in milliseconds) an item of the popup menu has to be selected before the submenu is shown. Note that the construction of a submenu might require a query to the ConceptBase server. If the option *"Popup Menu blocks while waiting for server"* is activated, then the editor will block the UI while it waits for an answer of the ConceptBase server. Otherwise, the query to the server will be executed in a separate thread, and interaction with the UI will be possible. If you have the problem that some submenus are still shown after you have used the popup menu, set the delay to 0 and activate the option *"Popup Menu blocks while waiting for server"*.

View menu

The graph editor has an experimental layout algorithm that may be useful to reorganize the layout of a complex graph. The heuristic of the layout algorithm is rather simple and does not minimize link crossings.

Enable automatic layout: Call a layout algorithm everytime the graph is changed, e.g. by expanding the attributes. Disabled by default. Enabling it is not recommended.

Undo last layout operation: Undo the last change of the graphical layout. This option is only available when automatic layout is enabled.

Layout graph: Call the layout algorithm.

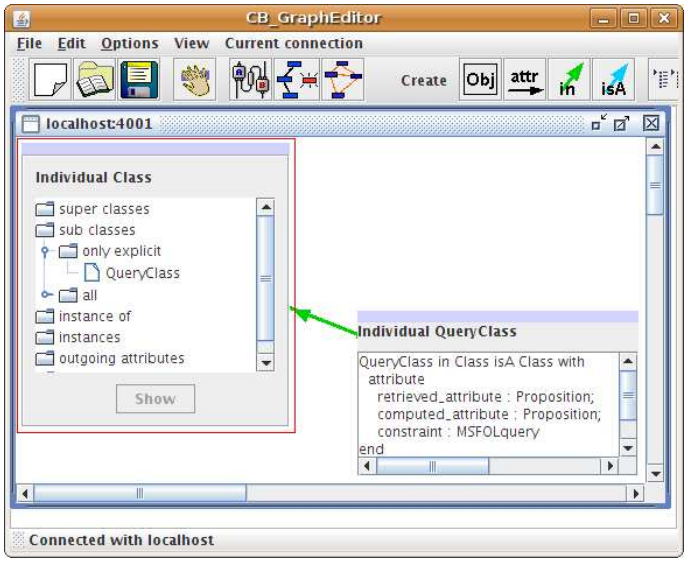

Figure 8.8: Component view of Telos objects: tree and frame

Zoom: Set the zoom factor of the graph, e.g. 120 (20 percent enlarged).

200,100,75,50,25: : Set the zoom factor accordingly.

Current connection menu

These operations have effect on the current connection, i.e. the connection of currently activated frame.

Query to server: This operation will open a dialog which prompts you to enter the name of a query (see figure 8.9). The query can also be parameterized. If you click on the "Submit Query" button, the query will be sent to the server and the result will be displayed in the listbox. You can select the objects that should be added to the graph. Selection of multiple objects is possible.

Validate and update shown objects: This operation will check for every object, if it is still valid (i.e. if it still exists in the database), update the graphical type of the object, and the internal cache of the object is deleted (see below, section 8.2.7). Depending on the option "Invalid Telos Objects" (see above), the objects will be either marked or removed from the current view.

Validate and update selected objects: Similar to the previous option but is only applied to objects in the graphical view that have been selected.

Change graphical palette: The current graphical palette (=assignment of graphical types to nodes and links) is replaced by another one.

8.2.4 Tool bar

The tool bar (see figure 8.10) consists of a set of buttons that are mainly short cuts for some menu items. The right half of the tool bar provides buttons for the creation of Telos objects.

Open a new frame (without connection): Opens a new frame without a connection to a server. Within this frame, you can create new Telos objects, load existing layouts and save new layouts. Information about the graphical types is loaded from an XML file included in the JAR file (`$CB_HOME/lib/classes/cb.jar`).

Load layout: see *File Menu* → *Load*

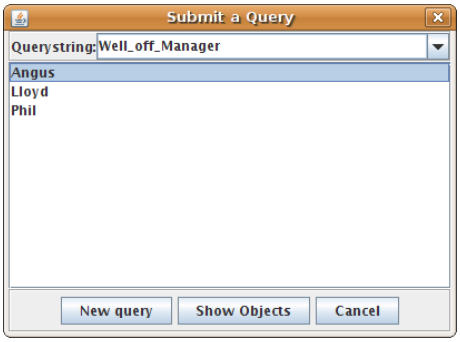

Figure 8.9: Query dialog

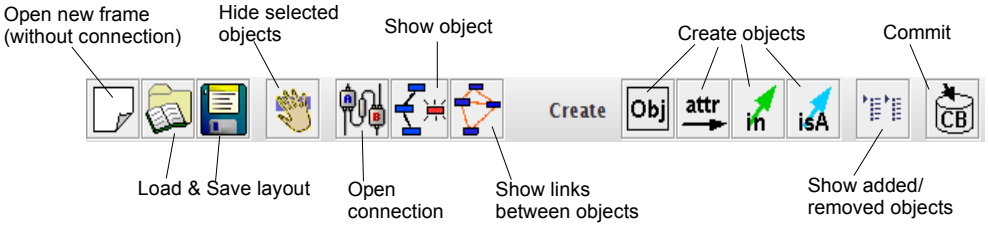

Figure 8.10: Tool Bar of the Graph Editor

Save layout: see *File Menu* → *Save*

Hide selected objects: Hides the selected objects from the current view. This operation has no effect on the current database, i.e. the objects will be not deleted from the database.

Open connection: Opens a new frame with a new connection to a server. See *File menu* → *Connect to server*.

Show object: This operation adds a new object to the graph. You will be prompted to enter the object name of the object you want to add to the graph.

Show links between marked objects: This operation will search for relationships between the selected objects. Do not select too many objects for this operation, n^2 queries have to be evaluated for n objects.

Creation of objects: The following four buttons open the "Create Object" dialog to create new individual objects, attributes, instantiations, and specializations. See section 8.2.6 for more details.

Show added/removed objects: Shows the objects that have been added or removed since the last commit (or since the window has been opened). Here, you can also select objects to undo the change, i.e. remove added objects or re-insert removed objects.

Commit: Sends the changes to the server. The list of objects to be added or removed is transformed into a set of Telos frames and transferred to the server.

8.2.5 Popup menu

The popup menu is activated by a click on the right mouse while the cursor is located over an object.

- **Toggle component view:**
  switches the view of this object. In the detailed component view, you can either see the frame of this object or tree-like representation of super- and subclasses, instances, classes, and attributes of this object (see figure 8.8).

- **Super classes, sub classes, classes, instances:**
  for each menu item you can select whether you want to see only the explicitly defined super classes (or sub classes, etc.) or all super classes including all implicit relationships. The query to the ConceptBase server to retrieve this information will be done when you select the menu item. So, the construction of the corresponding submenu might take a few seconds.

- **Incoming and outgoing attributes:**
  The Graph Editor will ask the ConceptBase server for the attribute classes that apply to this object. For each attribute class, it is possible to display only explicit attributes or all attributes as above. The attribute class "Attribute" applies for every object and all attributes are in this class. Therefore, all explicit attributes of an object will be visible in this category. However, there will be no attributes shown in the "All" submenu, as it would take too much time to compute the extension of all implicit attributes.

- **Add Instance, Class, SuperClass, SubClass, Attribute, Individual:**
  These menu items will open the "Create Object" dialog where you can specify new objects that should be created in the database. Note, that these modifications are not performed directly on the database. The editor will collect all modifications and send them to the ConceptBase server when you click on the "Commit" button. See section 8.2.6 for more details.

- **Delete object from database:**
  This operation will delete the object from the database. As for the insertion of objects before, the modification will be send to the server when you click on the "Commit" button. Note that this operation has an effect on the database in contrast to the next operation.

- **Hide object from view:**
  The object will be removed from the current view. This operation has no effect on the database, i.e. the object will not be deleted from the database.

- **Display in Workbench:**
  This operation will load the frame of the object into the Telos editor.

- **Show in new Frame:**
  A new internal window (within the Graph Editor) will be shown and the selected object will be shown in the new window.

8.2.6 Editing of Telos objects

The Graph Editor supports also the creation and deletion of Telos objects. A Telos object in the context of the Graph Editor is a *proposition* as described in chapter 2. As described there, there are four types of Telos objects:

- Individuals,
- Instantiations (InstanceOf),
- Specializations (IsA), and
- Attributes.

For each object type, we provide a dialog to create this object type as shown in figure 8.11. This dialog is opened by clicking on one of the "Create" buttons in the tool bar, or by selecting an "Add ..." item from the popup menu (e.g, "Add instance" or "Add subclass"). If there are some objects selected in the current

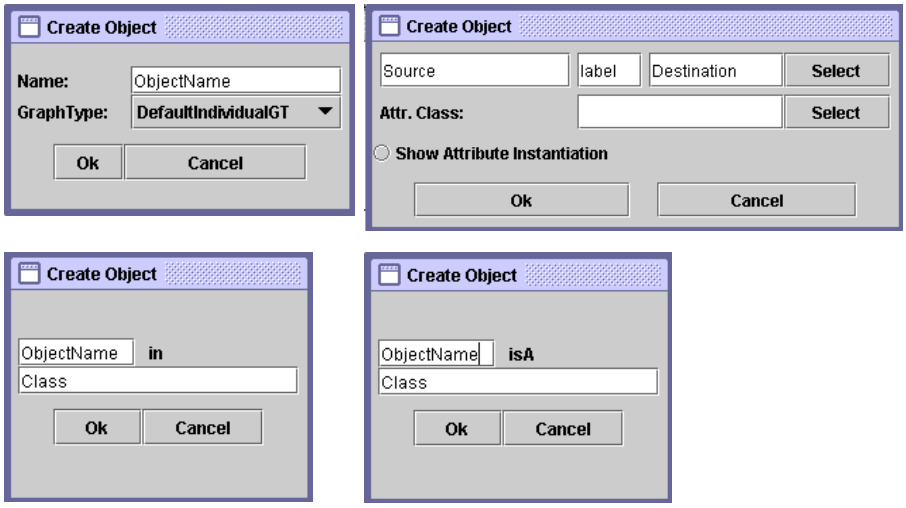

Figure 8.11: Create Object dialogs for Individuals, Attributes, Instantiations, and Specializations

frame, then the object names of these objects will be inserted into the text fields of the dialog in the order they have been selected (i.e., the first text field will contain the name of the object which has been selected first). Furthermore, if you move the cursor into a text field in the "Create Object" dialog and select an object in the graph, then the name of this object will be inserted into the text field.

As changes might lead to a temporary inconsistent state of the database, we do not execute the changes directly on the database. They are stored in an internal buffer in the Graph Editor and executed when you hit the "Commit" button in the tool bar.

Creating Individuals: If you want to create a new individual object, you just have to specify the object name. You have to enter a valid Telos object name, for example it must not contain spaces. In addition, you can select a graphical type for the object. Note that the selection of the graphical type has no effect in the database, e.g. by selecting the graphical type of a class (`ClassGT`) the object will not declared as an instance of `Class`. If you have performed the commit operation, the object will get the "correct" graphical type from the server.

Creating Instantiations: In the dialog for instantiations, you have to enter the name of the instance and the name of the class in the two text fields. If the object entered does not yet exist, it will be created and represented in the default graphical type.

Creating Specializations: This dialog is similar to the one before except that you specify here the name of the subclass and the name of the superclass. Objects that do not exist yet, will be created and represented in the default graphical type.

Creating Attributes: This is the most complex dialog as you have to specify the source, the label, the value, and the category of the attribute. The source and the value (or destination) of the attribute are normal object names. The label may be any valid Telos label. The attribute category has to be a select expression specifying an attribute category (e.g., `Employee!salary`, see chapter 2). The attribute category can be selected from a listbox by clicking on the "Select" button next to the text field of the attribute category. All attribute categories that apply to the current source of the attribute will be shown. Note that the list will be empty if the source object does not yet exist in the database.

If you have specified the attribute category, you can also select the attribute value from a listbox by clicking on the "Select" button next to the text field for the attribute value. The listbox will show all instances of the destination of the attribute category (e.g. all instances of `Department` for the category `Employee!dept`).

If you select the radio button "Show Attribute Instantiation" then the Graph Editor will also show the instantiation link for the attribute. For example, if you create a new attribute for `John` with the

label `JohnsDept` in the attribute category `Employee!dept`, then the instantiation link between `John!JohnsDept` and `Employee!dept` will also be shown. As the graph gets quite confusing with too many links, this radio button is not selected by default.

Deletion of objects is also possible. As this operation should not be mixed up with the removal of an object from the current view, this operation is just available from the popup menu (item "Delete Object from Database").

As you might make mistakes while editing the model, there is the possibility of undoing changes. The button "Show added/removed objects" list all objects that have been added or removed (since the last succesful commit or since the connection has been established). A screenshot of the dialog is shown in figure 8.12. The left list shows the objects that have been added, the right list shows the objects that have been removed. By clicking on the button "Re-Insert/Delete" object, the selected objects will be re-inserted in or deleted from the graph [7].

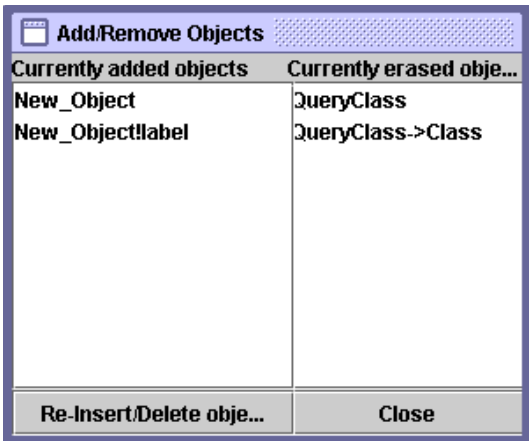

Figure 8.12: List of objects which have been added or removed

If you are satisfied with the changes you have done, you can click on the "Commit" button. Then, the Graph Editor will transform the objects to be added or removed into a list of Telos frames and send them to the server using the TELL, UNTELL, or RETELL operation. If the operation was successful, all *explicit* objects will be checked if they are still valid and if their graphical type has changed (as in the "Validate and update" operation from the "Current Connection" menu). If there is an error, the error messages of the server will be displayed in a message box. The internal buffer with the objects to add or remove will be not changed in this case.

8.2.7 Caching of query results within the graph editor

To improve the performance of the Graph Editor, several caches are used. On the other hand, the use of a cache causes several problems which will be addressed in this section. In particular, the caches of the Graph Editor are not updated automatically if the corresponding data in the server is updated.

Graphical Palette and Graphical Types: When a connection to a server is established, the Graph Editor loads the graphical palette and all its graphical types including their properties and other information. If an object has to be shown, the server sends only the name of the graphical type, the information about the properties are taken from the cache. Thus, if you change the graphical palette or a graphical type after the Graph Editor has established the connection, this change will not be visible in the Graph Editor. There is currently no method implemented to update the cache manually.

[7] You can unselect an object by holding down the Control key and clicking on the object.

Graphical Types of Objects: When an object is loaded from the server also the graphical type for this object is retrieved. The graphical type of the object is updated when you invoke the "Validate and Update" operation from the "Current Connection" menu.

Lists of super/sub classes, classes/instances, attributes: The lists in the popup menu or in the tree-like view of an object are produced by evaluating queries. To reduce the communication between client and server, each query will only be evaluated once (when the corresponding popup menu should be shown or when the part of the tree should be shown). The result will be stored in a cache for each object. This cache is emptied when you invoke the "Validate and Update" operation from the "Current Connection" menu.

8.3 An example session with ConceptBase

In this section we demonstrate the usage of the *ConceptBase User Interface*, by involving an example model. It consists of a few classes including `Employee, Department, Manager`. The class `Employee` has the attributes `name, salary, dept, and boss`. In order to create an instance of `Employee` one may specify the attributes `salary, name, and dept`. The attribute `boss` will be computed by the system using the `bossrule`. There is also a constraint which must be satisfied by all instances of the class `Employee` which specifies that no employee may earn more money than its boss. The Telos notation for this model is given in Appendix D.1.

8.3.1 Starting ConceptBase

To start a *ConceptBase* session, we use two terminal windows, one for the *ConceptBase* server and one for the usage environment. We start the *ConceptBase* server by typing the command

```
CBserver -p 4200 -d test
```

in a terminal window of, let us say machine alpha[8]. The parameter -p sets the port number under which the CBserver communicates to clients and the parameter -d specifies the name of the directory into which the CBserver internally stores objects persistently. Then, we start the usage environment with the command `CBjavaInterface` in the other window.

It is also possible to start the ConceptBase server from the user interface. To do so, choose "Start CBserver" from the "File Menu" of CBiva (see section 8.1.1) and specify the parameters in the dialog which will be shown (see figure 8.13). The option `Source Mode` controls whether the CBserver accessing the database via the `-d` parameter (database maintained in binary files), or via the `-db` parameter (database maintained both in binary files and in source files). See section 6 for more details. Once the information has been entered via the OK button, the server process will be started and its output will be captured in a window. This output window provides also a button stop the server. If you started the server this way then you can skip the next section, as the user interface will be connected to the server automatically.

8.3.2 Connecting CBiva to the ConceptBase server

Next we establish a connection between the *ConceptBase* server and the usage environment. This is done by choosing the option **Connect** from the File menu of the *ConceptBaseWorkbench*. An interaction window appears (see Figure 8.14) querying for the host name and the port number of the server (i.e. the number we have specified within the command `CBserver -p 4200 -d test`).

[8] A full list of all parameters is described in section 6. Note that the script `CBserver` must be in the search path. It is available in the subdirectory `bin` of your ConceptBase installation directory.

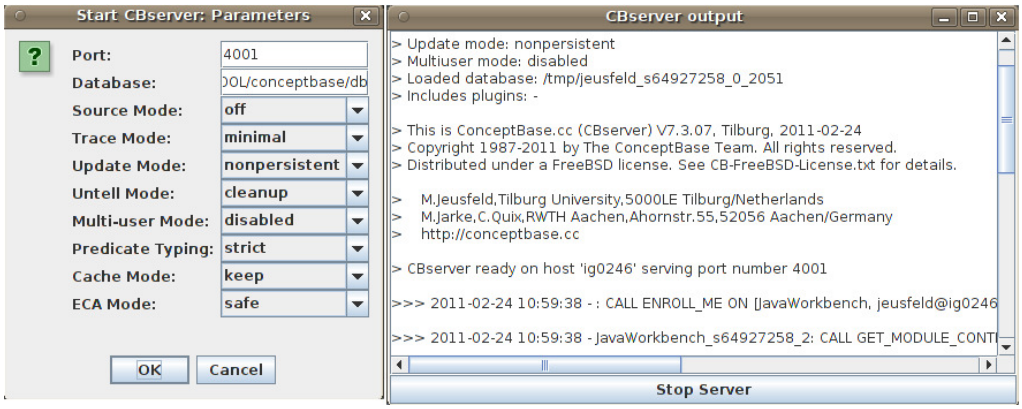

Figure 8.13: Start CBserver dialog and CBserver output window

Figure 8.14: The connect-to-server dialog

8.3.3 Loading objects from external files

The objects manipulated by *ConceptBase* are persistently stored in a collection of external files, which reside in a directory called **application** or **database**[9]. The actual directory name of the database is supplied as the -d parameter of the command `CBserver`.

The -u parameter of the `CBserver` specifies whether updates are made persistent or are just kept in system memory temporarily. Use `-u persistent` for a update persistence or `-u nonpersistent` for a non persistent update mode [10].

The database can be modified interactively using the editor commands TELL/UNTELL. Another way of extending ConceptBase databases is to load Telos objects (expressed in frame syntax) stored in plain text files with the extension *.sml. Call the menu item **Load Model** from the **File Menu** to add these objects to the database. In our example the database (directory) Employee was built interactively and can be found together with files containing the frames constituting the example in the directory

CB_HOME`/examples/QUERIES`

where you have to replace *CB_HOME* with the *ConceptBase* installation directory. The following files contain the objects of the Employee example expressed in frame syntax: `Employee_Classes.sml`, `Employee_Instances.sml`, `Employee_Queries.sml`. An alternative to interactively building a database is to start the server with an empty database (`-d` ⟨`newfile`⟩) and then add the objects in these files by using **Load Model**. Note, that the *.sml extension may be omitted. During the load operation of external models, *ConceptBase* checks for syntactical and semantical correctness and reports all errors to the history window as it is done when updating the object base interactively using the editor. This protocol field collects all operations and errors reported since the beginning of the session.

[9] Historically, we used the terms 'application' or 'object base' instead 'database'. We now believe that 'database' is a much better term.

[10] In nonpersistent update mode, the database is actually copied to a temporary directory. This copy will be removed when you shutdown the server.

8.3.4 Displaying objects

To display all instances of an object, e.g. the class `Employee`, we invoke the Display Instance facility by selecting the item **Display Instances** from the menu bar. In the interaction window we specify `Employee` as object name. The instances of the class `Employee` are then displayed (see Figure 8.15).

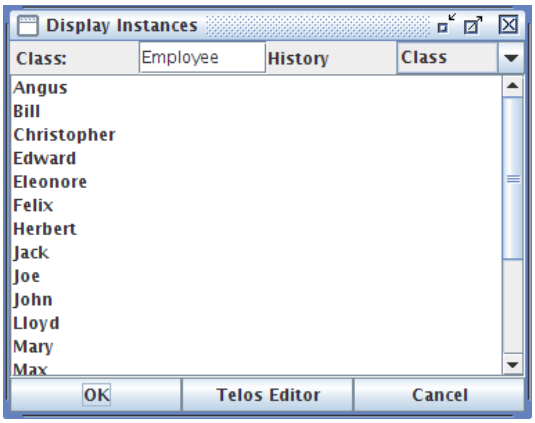

Figure 8.15: Display of `Employee` instances

After selecting a displayed instance we can load the frame representation of an instance to the Telos Editor or display further instances.

8.3.5 Browsing objects

The *Graph Editor* is the preferred tool for browsing the objects managed by a ConceptBase server. The Graph Editor is started by using the menu item "Graph Editor" from the "Browse" menu of CBiva. Select `Employee` as the initial object to be shown in the Graph Editor. After starting the Graph Editor, it will open an internal frame, connect it with the current server, and load the `Employee` object.

The *ConceptBase Graph Editor* (described in detail in section 8.2) allows you to display arbitrary objects from the current onceptBase server. Then, we select the `Employee` object and choose the **Sub classes** option from the context menu available via the right mouse-button. We choose to only display explicit subclasses from the submenu and select the `Manager` object. The displayed graph is now expanded (figure 8.16).

Now we expand the node `Manager`, a subclass of `Employee`, by choosing the menu item **Instances** from the popup menu for Manager. We select the menu item "Show all" to display all instances of `Manager`. The resulting graph is shown in figure 8.17.

Note that different object types are represented by different graphical objects. The instances of `Manager` are shown only as grey rectangles, because they are normal individual objects. The nodes `Manager`, `Salesman`, `Employee` etc. are shown as ovals, since these nodes are instances of the system class `SimpleClass` (see for a full description of graphical object semantics: Appendix C).

One can move nodes and links by selecting a node or a link and then holding down the left mouse button while moving the cursor to a different position. When the button is released the selected object will be located at the current position and the related links are redisplayed. Selection and movement of multiple nodes and links is also possible.

We can further experiment with the graph editor by showing the classes and attributes of Employee. The classes of Employee are shown by selecting "Instance of" from the popup menu. Attributes of an object can be shown by selecting "Outgoing attributes" from the menu. The next submenu will show all attribute classes that apply for the current object. In our example, `Employee` is an instance of `Class`. Therefore, it has the attribute classes `constraint`, `rule`, and `mrule` (see figure 8.18). The attribute class `Attribute` applies to all objects as in Telos any kind of object can have an attribute. Furthermore, all attributes of an object are member of the attribute class `Attribute`. As we want to see all attributes,

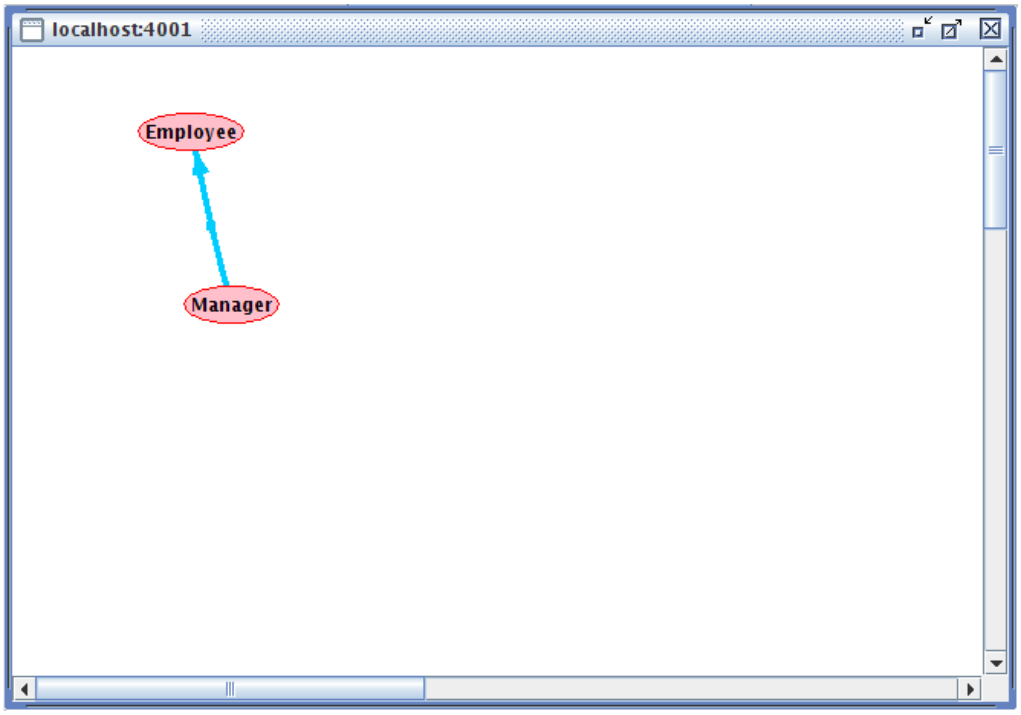

Figure 8.16: The resulting graph after expanding the node `Employee` with subclasses

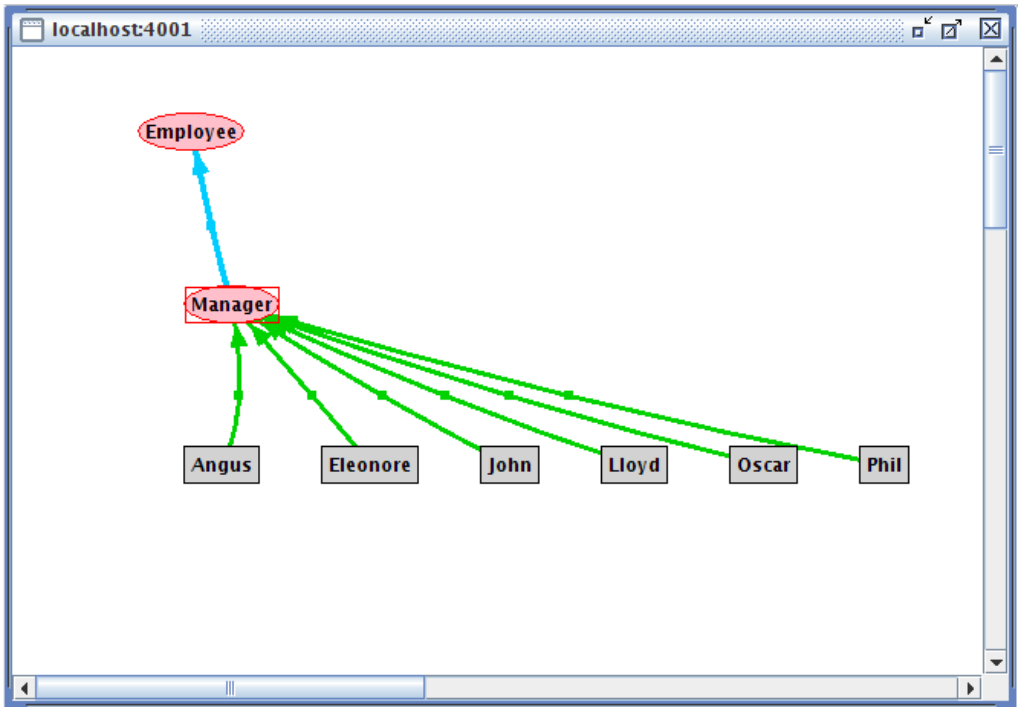

Figure 8.17: The resulting graph after expanding with the instances of `Manager`

we select this attribute class and select “Show all” from the next submenu. All attributes and their values will be shown in the Graph Editor.

The Graph Editor can also show implicit relationships between objects, e.g. relationships deduced by

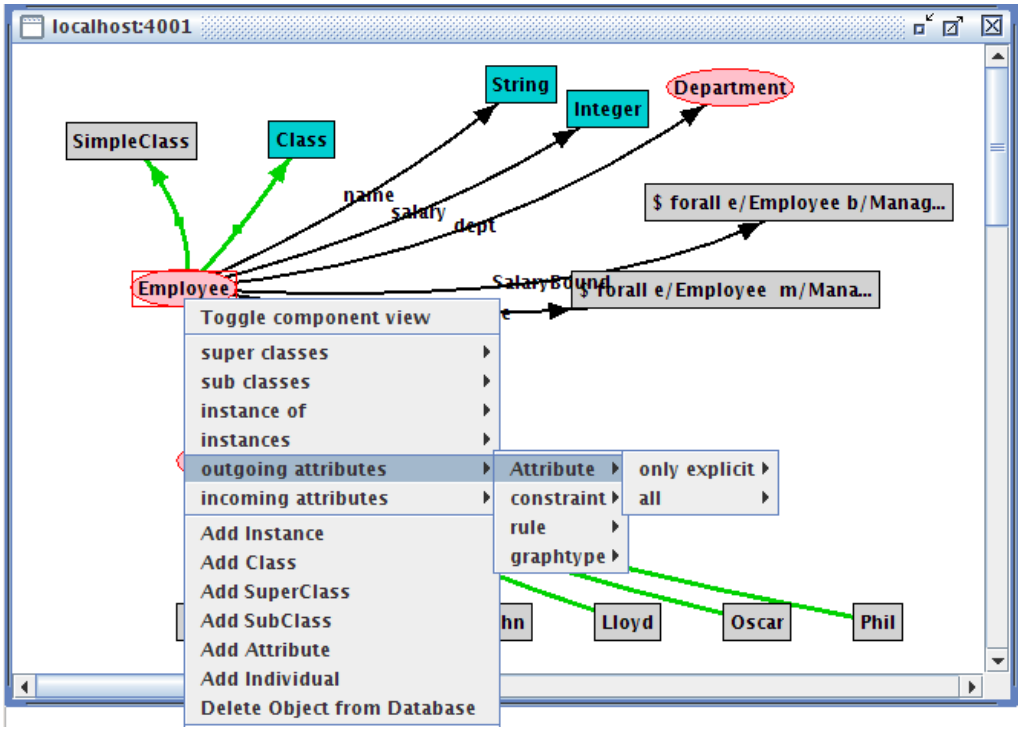

Figure 8.18: The graph after expanding it with the classes and the attributes of the class `Employee`

rules or the Telos axioms. For example, if we select the object `John` and select "Instance of" from the popup menu, we can display the implicit classes of `John` by selecting "All" from the next submenu. As `John` is an instance of `Manager` and `Manager` is a subclass of `Employee`, `John` is also an instance of `Employee`. As there is no explicit object `John->Employee`, the instantiation link between `John` and `Employee` will be represented as an implicit link, i.e. a dashed line (see figure 8.19).

The same applies also to attribute links. For example, the employee `Herbert` has an implicit `boss`-attribute to `Phil`. This can be shown by selecting "Outgoing attributes" → "boss" → "All" → "Phil" from the popup menu. Note, that the submenu "All" for the attribute class `Attribute` will be always empty as only explicit attributes can be displayed in this category.

8.3.6 Editing Telos objects

Editing Telos objects with the Telos editor

Before we are able to edit a Telos object, we have to load its frame representation in to the Telos Editor field first. For loading a Telos object to the Editor field, we choose the **Telos Editor** Button from either the Display Queries oder Display Instances Browsing facilities or the Load Frame button from the *ConceptBaseWorkbench* window (see Figure 8.20).

Now we add an additional attribute, e.g. `education`, to the class `Employee` (for the description of the Telos syntax see Appendix A). We have added the line `education : String` as shown in figure 8.21. To demonstrate error reports from the *ConceptBaseWorkbench* and how to correct them, we have made mistakes in the syntax notation of the added attribute.

By clicking the left mouse button on the **Tell** icon, the content of the editor is told to the *ConceptBase* server. Syntactical and semantical correctness is checked and the detected errors are reported to the Protocol field. The report resulting from our mistakes by specifying the new attribute is also shown in figure 8.21. Note, that this syntax error would have been already detected at the client side without interaction with the server if we would have enabled the option "Pre-parse Telos Frames" in the options menu.

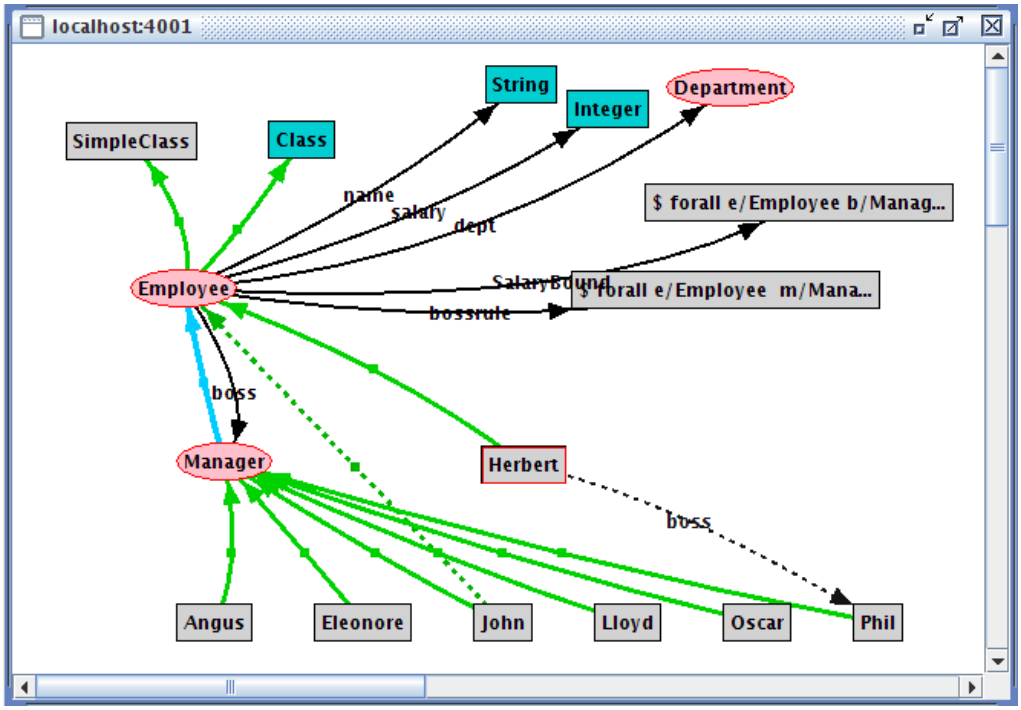

Figure 8.19: The graph showing implicit instantiation and attribute links

We correct the error by adding a semicolon to the previous line and choose the **Tell** symbol again. This time, since there are no further mistakes, the additional attribute is added to the class `Employee`.

Now we can choose again the item **Outgoing Attributes** from the popup of the *Graph Editor* window for the node Employee. If we select "Show all" attributes of the attribute class "Attribute" the new attribute will NOT be shown. The Graph Editor uses an internal cache which will only be updated on request. Therefore, we select the object `Employee` and select "Validate and update selected objects" from the menu "Current connection". The cache for the object `Employee` will be emptied. Now, showing all attributes should add the new attribute `education` to the graph.

Editing Telos objects using the graph editor

Telos objects can also be edited graphically in the Graph Editor. In our example, we want to add another attribute named `address` to the class `Employee`. The attribute destination of this attribute should be a new class called `Address`.

First, we select the object Employee and click on the "Create Attribute" button in the tool bar or select "Add Attribute" operation from the tool bar. As we have selected the Employee object, it should be already inserted as source of the attribute. We have to type the label (address) and the destination of the attribute in the text fields (see figure 8.22). As this attribute does not belong to a specific attribute category (it is just an attribute), we do not have to specify an attribute class.

By clicking on the Ok button, the Graph Editor will create a new object for `Address` represented by the default graphical type (a gray box). Then, it will create the attribute link from `Employee` to `Address` with the label `address`. The result is shown in figure 8.23.

Now, we want to declare `Address` as an instance of `Class`. Therefore, we select `Address`, hold down the Shift-key and select `Class`. Both objects should be selected now. We click on the "Create Instantiation" button and a dialog as shown in figure 8.23 should appear.

As we have selected the objects in the correct way, the dialog already specifies the object we want to create (`Address` in `Class`) and we can click directly on Ok. The new instantiation link will be added to the graph.

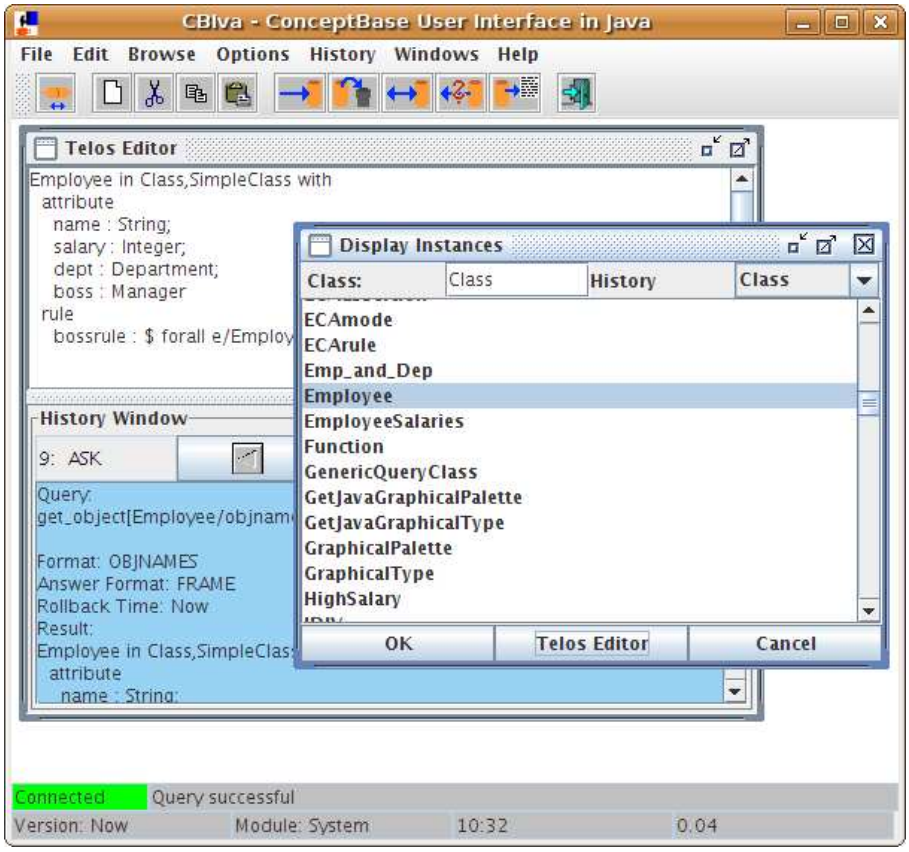

Figure 8.20: The TelosEditor Field with the object `Employee`

Now, we are satisfied with our changes and want to commit them in the server. So far, the changes have been stored in an internal buffer of the Graph Editor and have not been sent to the server. We click on the "Commit" button in the upper right corner. The Graph Editor generates now Telos frames for the added objects and sends them to the server. If we did not make an error, all changes should be consistent and accepted by the server. This is shown by the appearing message box "Changes committed". If an error occurs, an error message will displayed instead. The graphical editing is an alternative to the textual editing via the Telos Editor. It is appropriate for incremental changes to a model. Larger changes should better be made via the Telos Editor or even to an external text file that is loaded via the *File / Load Model* facility of CBiva. If the changes were successfully told to the ConceptBase server, the Graph Editor reloads the information of every visible object. In particular, the graphical types of the objects will be updated. As the object `Address` is declared as instance of `Class`, it will get the graphical type of a class, i.e. a turquoise box. The result is shown in figure 8.24. The object `Employee` is shown in the detailed component view, in this case the frame representation of the object is shown. As you can see, the attribute `address` is now also visible in the frame representation.

8.3.7 Using the query facility

Lets assume that we need to ask the server for all `Employees` working for `Angus`. We open a new Telos Editor (see menu item Browse). Then, we define a new query class (`AngusEmployees`) as follows:

```
AngusEmployees in QueryClass isA Employee with
constraint
    c: $ (this boss Angus) $
end
```

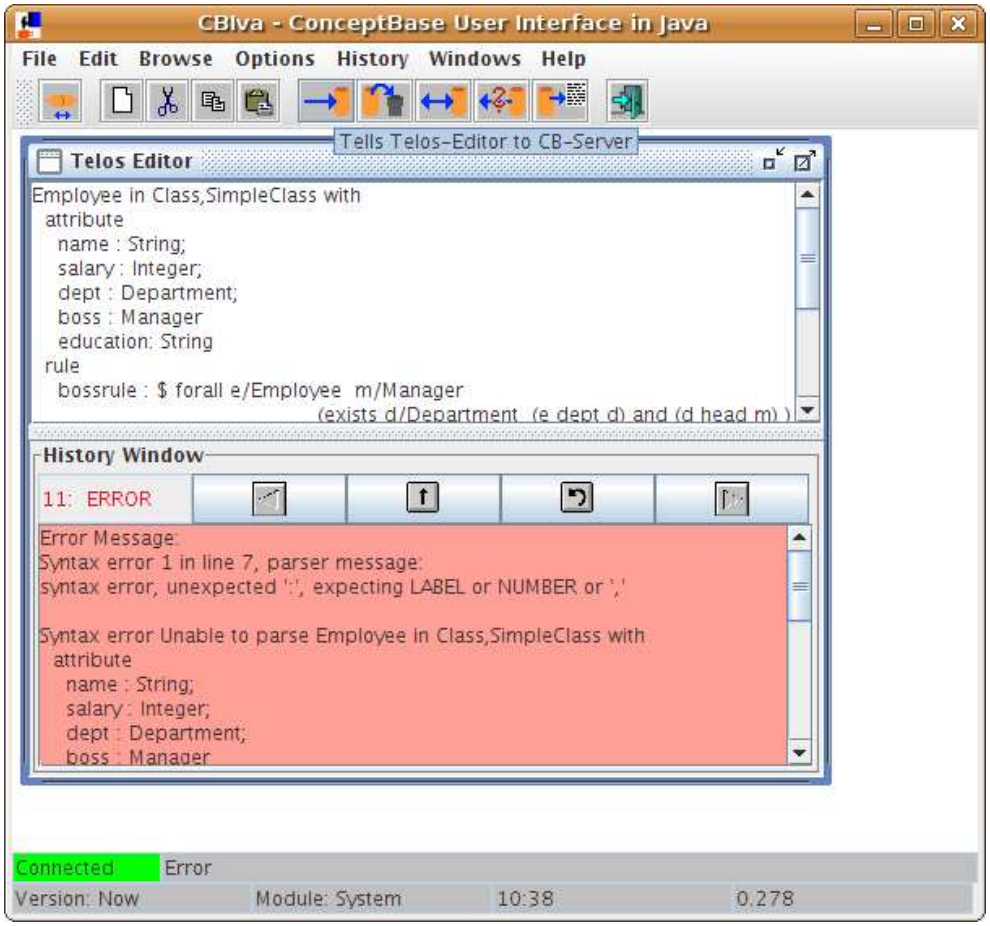

Figure 8.21: Trying to add an attribute to the class `Employee` with the resulting error report

We can tell this query, so that it is stored in ConceptBase and we can reuse it later, or we can just *ask* the query, i.e. the query will told temporarily and evaluated. If we ask the query, the answer is displayed in the Telos Editor field as well as in the history window. Figure 8.25 shows the *ConceptBaseWorkbench* containing the query class and the answer.

We can also execute this query from the Graph Editor. From the menu "Current connection" we select "Query to server". A dialog we ask for the name of query class. If we have told the example query, we can type `AngusEmployees` in the text field and hit on the "Submit Query". The query will be evaluated and the objects in the result will be shown in the list box. We can select the objects which should be added to the graph (multiple selection with the Shift-key is possible) and click on the "Show objects" button. The selected objects will be added to the graph, however with no connection to existing objects.

8.4 Usage as applet

Both applications, CBiva and CB Graph Editor, can be used as an applet within a web browser with a plugin for JDK 1.4. To start the applications as applet, you have to specify inside the applet class of the program. For CBiva, the applet class is

```
i5.cb.workbench.CBIvaApplet.class
```

and for the CB Graph Editor, the main applet class is

```
i5.cb.graph.cbeditor.CBEditorApplet.class
```

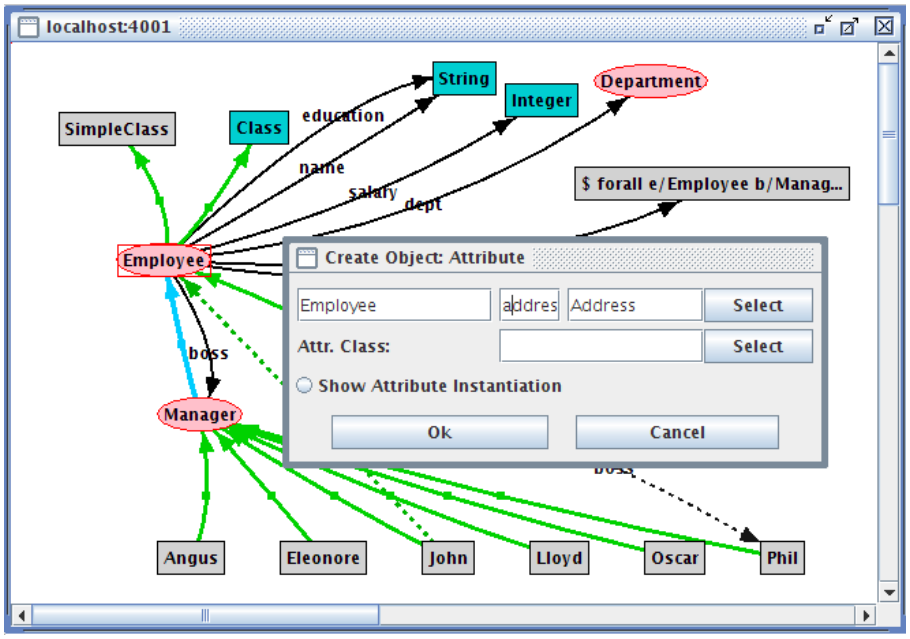

Figure 8.22: Adding an attribute to `Employee`

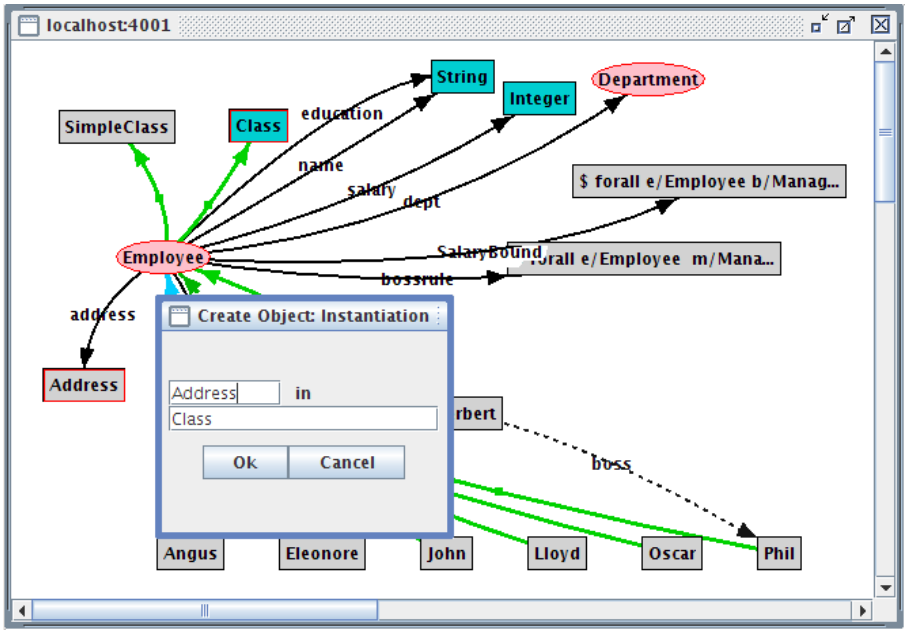

Figure 8.23: Adding an instantiation link between `Address` and `Class`

For example, to include the CB Graph Editor as applet inside your web page, use the following applet tag:

```
<applet archive="cb.jar,jgl3.1.0.jar"
   code="i5.cb.graph.cbeditor.CBEditorApplet.class"
   width=10 height=10>
</applet>
```

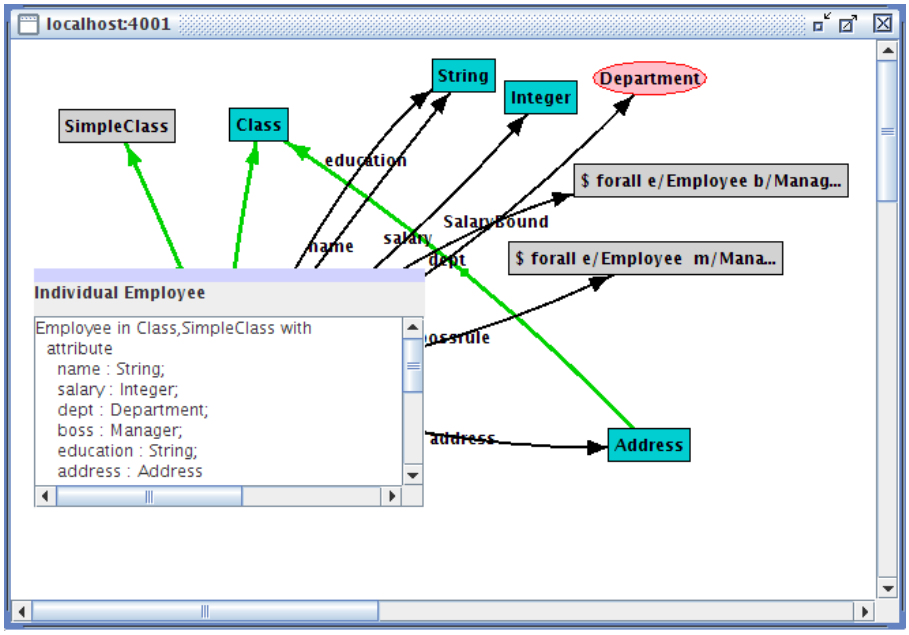

Figure 8.24: The resulting graph after commit

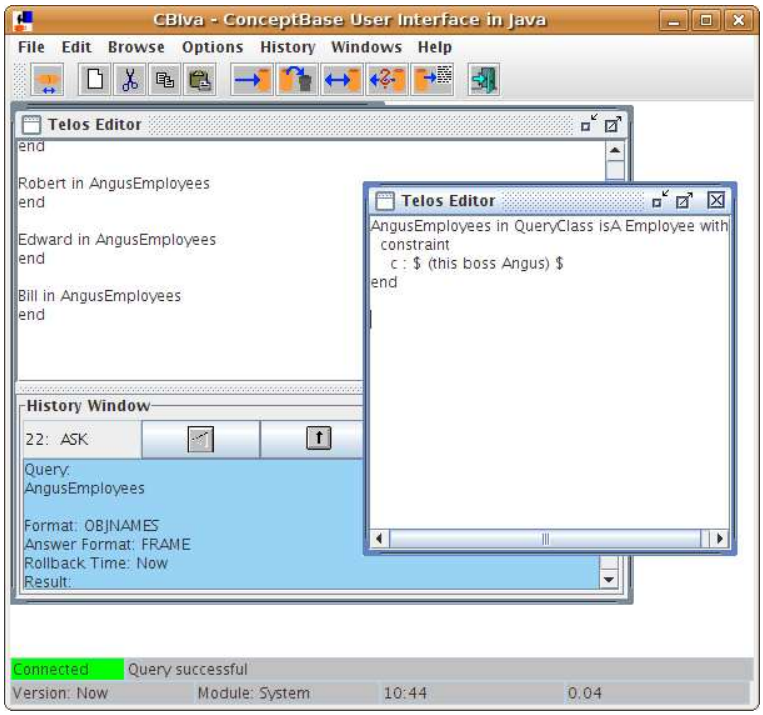

Figure 8.25: Query class and its answer

The `archive` parameter specifies the names of the archives to be loaded, the `code` parameter is the name of the applet class to be startet. As we have not specified a `codebase` parameter, the JAR files have to be located in the same directory as the HTML file. See `http://java.sun.com/j2se/1.4/docs/guide/misc/applet.html` for more information on the applet tag.

8.5 Configuration file

The configuration options are stored in a file ".CBjavaInterface" in the home directory of the user (`~/` on UNIX or "`Documents and Settings`" on Windows). The settings are stored automatically on exit. You can edit the file manually with a normal text editor. It contains name-value pairs in the format `name=value`. All options can also be modified by the user interface, so it is not necessary to edit this file.

- Options related to CBiva

  PathForLoadModel: Path used by the load model dialog (contains the most recent directory selected in a dialog).

  RecentConnections: Comma-separated list of recent connections in the format host/port, applies also to the graph editor.

  PreParseTelosFrames: Frames are be parsed on client-side before sent to the server (true/false).

  UseQueryResultWindow: Use the query result window to display results of a query (true/false).

  ConnectionTimeout: Number of milliseconds the interface waits for a response of the server

  LPICall: Enable LPI-Call (internal use only).

- Options related to CB Graph Editor

  PathForLayout: Path used by the dialog to store and load layouts (contains the most recent directory selected in a dialog).

  ComponentView: Default view for the detailed representation in the graph editor (might be "`frame`" or "`tree`").

  SaveLayoutWithGraphType: Layouts of the graph editor are stored with all information about graphical types (true/false).

  InvalidObjectsMethod: Specifies whether objects that have been identified as invalid should be marked or deleted (mark/delete).

  DiagramDesktopBackgroundColor: Background color of the desktop of the graph editor (comma-separated representation of an RGB-value).

  DebugLevel: Level for debug messages. Possible values are SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST (according to the java.util.logging package). Default is WARNING.

Bibliography

[NOTE!!] **A complete list of papers related to ConceptBase can be found at** http://www-i5.informatik.rwth-aachen.de/CBdoc/cblit.html .

[BBD*90] Bellosta, M.J., Bessede, A., Darrieumerlou, C., Gruber, O., Pucheral, P., Thevenin, J.M., Steffen, H.: GEODE: Concepts and Facilities. Manual, INRIA, Roquencourt, France, 1990.

[BBMR89] Borgida, A., Brachman, R.J., McGuiness, D.L., Resnick, L.A.: CLASSIC: a structural data model for objects. Proc. ACM-SIGMOD Intl. Conf. Management of Data, Portland, Or, 58-67, 1989.

[BCG*87] Banerjee, J., Chou, H.-T., Garza, J.F., Kim, W., Woelk, D., Ballou, N., Kim, H.-J.: Data model issues for object-oriened applications. ACM Trans. Office Information Systems 5, 1, 3-26, 1987.

[BMS84] Brodie M.L., Mylopoulos J., Schmidt J.W. (ed.): *On Conceptual Modeling*, Springer-Verlag, 1984.

[Chen76] Chen, P. P.-C.: The Entity-Relationship-Model – Towards a Unified View of Data. ACM Transact. on Database Systems. Vol. 1, No. 1, March 1976, pp. 9-36, 1976.

[CW96] Chen, W., Warren, D.S. : Tabled Evaluation with Delaying for General Logic Programs. Vol. 43, No. 1, January 1996, pp. 20-74, 1996.

[Eber97] Eberlein, A.: *Requirements Acquisition and Specification for Telecommunication Services*. PhD Thesis, University of Wales, Swansea, UK, 1997.

[Gall90] Gallersdörfer, R.: *Realization of a Deductive Object Base by Abstract Data Types* (in German), Diploma thesis, Universität Passau, Germany, 1990.

[HJEK90] Hahn, U., Jarke, M., Eherer, S., Kreplin, K.: CoAUTHOR: a hypermedia group authoring environment. In Benford, J.M., Bowers, S.D. (eds.): Studies in Computer-Supported Cooperative Work, North-Holland, 79-100, 1990.

[Jark93] Jarke, M. (ed.): *Database Application Engineering with DAIDA*, Springer-Verlag, 1993.

[JeJa91] Jeusfeld, M.A., Jarke, M.: From Relational to Object-Oriented Integrity Simplification. *Proc. 2nd Intl. Conf. Deductive and Object-Oriented Databases*, Munich, Dec. 1991; also as *Aachener Informatik-Berichte* 91-19, RWTH Aachen, Germany, 1991.

[JeSt92] Jeusfeld,M., Staudt,M.: Query optimization in deductive object bases. To appear in Freytag, Vossen, Maier (eds.): Query Processing for Advanced Database Applications, Morgan-Kaufmann, 1992; also as *Aachener Informatik-Berichte* 91-26, RWTH Aachen, Germany.

[Jeus92] Jeusfeld, M.A.: Änderungskontrolle in deduktiven Objektbanken. Dissertation, Universität Passau, Germany.

[JGJ*95] Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S.: ConceptBase: A deductive object base for meta data management. *Journal of Intelligent Information Systems*, Vol. 4, No. 2, pp. 167-192, March 1995.

[JJM*09] Jeusfeld, M.A., Jarke, M., Mylopoulos, J. (eds.): Metamodeling for Method Engineering. Cambridge, MA, 2009. The MIT Press, ISBN-10: 0-262-10108-4.

[JJN*98] Jeusfeld, M.A., Jarke, M., Nissen, H.W., Staudt, M.: ConceptBase - Managing Conceptual Models about Information Systems. In Bernus, P., Mertins, K., Schmidt, G., (eds.): Handbook on Architectures of Information Systems, Springer-Verlag, pp. 265-285, 1998.

[JJQV99] Jarke, M., Jeusfeld, M.A., Quix, C., Vassiliadis, P.: Architecture and Quality for Data Warehouses: An Extended Repository Approach *Information Systems*, Vol. 24, No. 3, pp. 229-253, 1999.

[JJR88] Jarke, M., Jeusfeld, M., Rose, T.: A KBMS for database software evolution: documentation of first ConceptBase prototype, Report MIP-8819, Universit"at Passau, Germany, 1988.

[JJR90] Jarke, M., Jeusfeld, M., Rose, T.: A software process data model for knowledge engineering in information systems. *Information Systems* 15, 1, 85-116, 1990.

[JJS*99] Jeusfeld, M.A., Jarke, M., Staudt, M., Quix, C., List, T.: Application Experience with a Repository System for Information Systems Development. In R. Kasckek (ed.): *Proceedings Entwicklungsmethoden fr Informationssysteme und deren Anwendung (EMISA'99)*. Reihe Wirtschaftsinformatik, Teubner Verlag, Stuttgart, Germany, pp. 147-174, 1999

[JMSV90] Jarke, M., Mylopoulos, J., Schmidt, J.W., Vassiliou, Y: Information systems development as knowledge engineering: a review of the DAIDA project. *Programirovanie* Vol. 17, No. 1, Academy of Sciences, Russia; also appeared as technical report MIP-9010, Universität Passau, Germany, 1990.

[JQC*00] Jarke, M., Quix, C., Calvanese, D., Lenzerini, M., Franconi, E., Ligoudistianos, S., Vassiliadis, P., Vassiliou, Y.: Concept Based Design of Data Warehouses: The DWQ Demonstrators. *ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*, Dallas, TX, May 2000. http://www-i5.informatik.rwth-aachen.de/lehrstuhl/projects/dwq/dwqdemo/

[JQJ98] Jeusfeld, M.A., Quix, C., Jarke, M.: Design and Analysis of Quality Information for Data Warehouses *17th International Conference on the Entity Relationship Approach (ER'98)*, Singapore, 1998.

[MaPr99] MasterProLog 4.1, Reference Manual. IT Masters, 1999.

[MBJK90] Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: a language for representing knowledge about information systems. *ACM Trans. Information Systems* Vol. 8, No. 4, 1990.

[NJJ*96] Nissen, H.W., Jeusfeld, M.A., Jarke, M., Zemanek, G.V., Huber, H.: Managing Multiple Requirements Perspectives with Metamodels IEEE Software, pages 37-48, March 1996.

[Quix96] Quix, C.: Sichtenwartung und Änderungsnotifikation für Anwendungsprogramme in deduktiven Objektbanken. Diploma Thesis, RWTH Aachen, 1996.

[QSJ02] Quix, C., Schoop, M., Jeusfeld, M.A.: Business Data Management for Business-to-Business Electronic Commerce *SIGMOD Record*, Vol. 31, No. 1, pp. 49-54, March 2002.

[RaDh92] Ramesh,B., Dhar,V.: Process knowledge-based group support for requirements engineering. *Journal Intelligent Information Systems* Vol. 1, No. 1, 1992.

[RJG*91] Rose, T., Jarke, M., Gocek, M., Maltzahn, C., Nissen, H.: A decision-based configuration process environment. *Software Engineering Journal*, Special Issue on Software Environments and Factories, July 1991.

[SIGA91] Rich, C. (ed.): "Special Issue on Implemented Knowledge Representation and Reasoning Systems", *SIGART Bulletin* Vol. 2, No. 3, June 1991.

[StJa96] Staudt, M., Jarke, M.: Incremental Maintenance of Externally Materialized Views. In *Proc. of the 22th VLDB Conference*, Bombay, India, 1996.

[Soir96] Soiron, R.: Kostenbasierte Anfrageoptimierung in deduktiven Objektbanken. Diploma Thesis, RWTH Aachen, 1996.

[Stan86] Stanley M.T.: *CML – a knowledge representation language with application to requirements modeling*, M.S.thesis, University of Toronto, Ontario, 1986.

[Stau90] Staudt, M.: Anfragereprd̂sentation und -auswertung in deduktiven Objektbanken. Diploma thesis, Universitd̂t Passau, 1990.

[TKDE90] Stonebraker M. (ed.): Special Issue on Database Prototype Systems, *IEEE Trans. on Knowledge and Data Engineering* Vol. 2, No. 1, March 1990.

[Wing90] Wing, J.M.: A Specifier's Introduction to Formal Methods. *IEEE Computer*, No. 9, pp 8-24, Sept. 1990.

Appendix A

Syntax Specifications

A.1 Syntax specifications for Telos frames

```
<object>        --> <objectname> <objectname> <inspec> <isaspec>
                      <withspec> <endspec>
                  | <objectname> <inspec> <isaspec> <withspec> <endspec>

<objectname>    --> ( <objectname>  )
                  |  <label> <bindings>
                  |  <objectname>  SELECTOR1 <label>
                  |  <objectname>  SELECTOR2 <objectname>

<bindings>      -->  <empty>
                  |  [ <bindinglist>  ]

<bindinglist>   -->  <singlebinding>
                  |  <bindinglist>  , <singlebinding>

<singlebinding> -->  <objectname> / <label>
                  |  <label> : <objectname>

<inspec>        -->  <empty>
                  |  in <classlist>

<isaspec>       -->  <empty>
                  |  isA <classlist>

<classlist>     -->  <objectname>
                  |  <objectname>  , <classlist>

<withspec>      -->  <empty>
                  |  with <decllist>

<decllist>      -->  <empty>
                  |  <declaration>
                  |  <decllist> <declaration>

<declaration>   -->  <attrcatlist> <proplist>

<attrcatlist>   -->  <label>
                  |  <attrcatlist>  , <label>

<proplist>      -->  <property>
```

```
                 |   <proplist>  ; <property>

<property>      -->  <label>  : <objectname>
                 |   <label>  : <complexref>
                 |   <label>  : <enumeration>
                 |   <label>  : <pathexpression>

<complexref>    -->  <objectname> <withspec> <endspec>

<enumeration>   -->  [ <classlist>  ]

<pathexpression>-->  <objectname>  SELECTORB <pathargument>

<pathargument>  -->  <label>
                 |   <label>  SELECTORB <pathargument>
                 |   <restriction>
                 |   <restriction>  SELECTORB <pathargument>

<restriction>   -->  ( <label>  : <enumeration>  )
                 |   ( <label>  : <pathexpression>  )
                 |   ( <label>  : <objectname>  )

<endspec>       -->  end

<label>         -->  ALPHANUM
                 |   LABEL
                 |   NUMBER
```

Note: ConceptBase represents internally object identifiers as *id_NUMBER* where *NUMBER* is a sequence of digits. For this reason, labels matching this pattern are forbidden in the Telos frame syntax.

A.2 Syntax of the rule and constraint language

In the definitions below the term `literal` is a synonym for predicate.

```
<assertion>     --> <rule>
                 |  <constraint>

<rule>          --> forall <variableBindList> ( <formula> ) ==> <literal>
                 |  <formula> ==> <literal>
                 |  <literal>

<constraint>    --> <formula>

<formula>       --> exists <variableBindList> <formula>
                 |  forall <variableBindList> <formula>
                 |  not <formula>
                 |  <formula> <==> <formula>
                 |  <formula> ==> <formula>
                 |  <formula> and <formula>
                 |  <formula> or <formula>
                 |  ( <formula> )
                 |  <literal>
                 |  <literal2>

<variableBindList>--> <variableBind> <variableBindList>
                   |  <variableBind>
```

```
<variableBind>  --> <varList> / <objectname>
                 |  <varList> / [ <objList> ]
                 |  ALPHANUM / <selectExpB>

<varList>       --> ALPHANUM , <varList>
                 |  ALPHANUM

<objectname>    --> <label>
                 |  <selectExpA>
                 |  <deriveExp>

<label>         --> ALPHANUM
                 |  LABEL
                 |  NUMBER

<literal>       --> FUNCTOR ( <literalArgList> )
                 |  ( <literalArg> <infixSymbol> <literalArg> )
                 |  ( <arExpr> COMPSYMBOL <arExpr> )
                 |  ( <literalArg> <label>/<label> <literalArg> )
                 |  BOOLEAN

<literal2>      --> ( <label> in <selectExpB> )
                 |  ( <selectExpA> in <selectExpB> )
                 |  ( <selectExpB> isA <selectExpB> )
                 |  ( <selectExpB> = <selectExpB> )

<infixSymbol>   --> INFIXSYMBOL
                 |  <label>

<literalArgList>--> <literalArg> , <literalArgList>
                 |  <literalArg>

<literalArg>    --> <objectname>

<arExpr>        --> <arExpr> + <arTerm>
                 |  <arExpr> - <arTerm>
                 |  <arTerm>

<arTerm>         --> <arTerm> * <arFactor>
                 | <arTerm> / <arFactor>
                 | <arFactor>

<arFactor>       --> ( <arExpr> )
                 | <objectname>
                 | <funExpr>

<selectExpA>    --> <selectExpA> <selector> <selectExpA>
                 |  ( <selectExpA> )
                 |  <label>

<selector>      --> SELECTOR1
                 |  SELECTOR2

<deriveExp>     --> <label> [ <deriveExpList> ]
                 | <label [ <plainArgList> ]
                 |  <funExpr>
```

```
<funExpr>        --> <label>()
                 |  <label>(<plainArgList>)

<deriveExpList> --> <singleExp> , <deriveExpList>
                 |  <singleExp>

<singleExp>     --> <literalArg> / <label>
                 |  <label> : <label>

<plainArgList> --> <literalArg> , <plainArgList>
                 |  <literalArg>

<selectExpB>    --> <label> SELECTORB <label>
                 |  <label> SELECTORB <selectExpB2>
                 |  <label> SELECTORB <restriction>

<selectExpB2>   --> <selectExpB>
                 |  <restriction> SELECTORB <label>
                 |  <restriction> SELECTORB <selectExpB2>
                 |  <restriction> SELECTORB <restriction>

<restriction>   --> ( <label> : <label> )
                 |  ( <label> : <selectExpA> )
                 |  ( <label> : <selectExpB> )
                 |  ( <label> : [ <objList> ] )

<objList>       --> <objectname> , <objList>
                 |  <objectname>
```

A.3 Syntax of active rules

The event, condition and actions of an ECArule are specified as a special assertion. Therefore, the syntax is an extension of the *normal* assertion language, shown in the section before.

```
<ecarule>       --> <variableBindList> ON <ecaevent> IF <ecacondition>
                    DO <actionlist> <optelseaction>

<ecaevent>      --> <eventop>(<literal>)
                 |  <eventop> <literal>
                 |  Ask(<literalArg>)
                 |  Ask <literalArg>

<eventop>       --> Tell
                 |  Untell

<ecacondition>  --> <condformula>
                 |  true
                 |  false

<condformula>   --> <literal>
                |   not <condformula>
                |   <condformula> and <condformula>
                |   <condformula> or <condformula>
                |   ( <condformula> )

<actionlist>    --> <action> , <actionlist>
                 |  <action>
```

```
<action>         --> <actionop>(<literal>)
                  |  <actionop> <literal>
                  |  noop
                  |  reject

<actionop>       --> Tell
                  |  Untell
                  |  Retell
                  |  Ask
                  |  CALL

<optelseaction> --> ELSE <actionlist>
                  |  <empty>
```

A.4 Terminal symbols

```
 ALPHANUM        --> [a-zA-Z0-9]+

 LABEL           --> everything except .|'"$:;!^->=,()[]{}/, newlines, tabs, ...
                 |   everything enclosed in " except " and \,
                       which must be escaped with \
                 |   everything enclosed in $ except $ and \,
                       which must be escaped with \

 NUMBER          --> REAL
                 |   INTEGER

 REAL            --> [-]?([0-9]+\.[0-9]*|[0-9]*\.[0-9]+)([Ee][-+]?[0-9]+)?

 INTEGER         --> [-]?[0-9]+

 BOOLEAN         --> TRUE
                 |   FALSE

 FUNCTOR         --> From | To | A | Ai | AL | In
                 | Isa | Label | P |  LT | GT | LE | GE | EQ | NE | IDENTICAL
                 | UNIFIES | Known

 COMPSYMBOL     --> < | > | <= | >= | = | <> | == | \=

 INFIXSYMBOL     --> COMPSYMBOL | in | isA

 SELECTOR1       -->  "!" | "^"

 SELECTOR2       -->  "->"| "=>"

 SELECTORB       -->  "." | "|"
```

A.5 Syntax specifications for SML fragments

This format is only internally used to represent Telos frames as Prolog terms. It is included only for historical reasons.

```
 <SMLfragment>  --> SMLfragment(<what> , <in_omega> , <in> , <isa> , <with> )

 <what>         --> what(<object> )
```

```
<in_omega>      --> in_omega(nil)
                 |  in_omega([<classlist> ])

<in>            --> in(nil)
                 |  in([<classlist> ])

<isa>           --> isa(nil)
                 |  isa([<classlist> ])

<with>          --> with(nil)
                 |  with([<attrdecllist> ])

<classlist>     --> class(<object> )
                 |  <classlist> , class(<object> )

<attrdecllist> --> attrdecl(<attrcategorylist> , <propertylist> )
        |  <attrdecllist> , attrdecl(<attrcategorylist> , <propertylist> )

<attrcategorylist>--> nil
                 |  [ <labellist> ]

<propertylist> --> nil
                 |  [ <propertylist2> ]

<propertylist2>--> property(<label> , <propertyvalue> )
                 |  <propertylist2> , property(<label> , <propertyvalue> )

<propertyvalue>--> <object>
                 |  <selectExpB>
                 |  enumeration( [ <classlist> ] )
                 |  [ <SMLfragment> ]

<selectExpB>    --> selectExpB( <restriction> , <selectOperator> , <selectExpB> )
                 |  selectExpB( <restriction> , <selectOperator> , <object> )
                 |  selectExpB( <object> , <selectOperator> , <selectExpB> )
                 |  selectExpB( <object> , <selectOperator> , <object> )

<restriction>   --> restriction( <label> , <selectExpB> )
                 |  restriction( <label> , enumeration( [ <classlist> ] ) )
                 |  restriction( <label> , <object> )

<selectOperator> --> dot
                 | bar

<labellist>     --> <label>
                 |  <labellist> , <label>

<label>         --> ALPHANUM
                 |  LABEL
                 |  NUMBER

<object>        --> derive([ <substlist> ])
                 |  <selectexp>

<substlist>     --> <singlesubst>
                 |  <substlist> , <singlesubst>
```

```
<singlesubst>  --> substitute(<object> , <label> )
                |  specialize(<label> , <label> )

<selectexp>    --> <label>
                |  select(<selectexp> , SELECTOR1, <label> )
                |  select(<selectexp> , SELECTOR2, <selectexp> )
```

Appendix B

O-Telos Axioms

O-Telos is the variant of Telos (originally defined by John Mylopoulos, Alex Borgida, Manolis Koubarakis and others) that is used by the ConceptBase system. This list is the complete set of pre-defined axioms of O-Telos and thus defines the semantics of a O-Telos database (without user-defined rules and constraints). The subsequent axioms are written in a first-order logic syntax but all can be converted to Datalog with negation (though there is some choice in the conversion wrt. mapping to rules or constraints).

- Axiom 1: Object identifiers are unique.
  $\forall\, o, x_1, l_1, y_1, x_2, l_2, y_2\ P(o, x_1, l_1, y_1) \wedge P(o, x_2, l_2, y_2) \Rightarrow$
  $(x_1 = x_2) \wedge (l_1 = l_2) \wedge (y_1 = y_2)$

- Axiom 2: The name of individual objects is unique.
  $\forall\, o_1, o_2, l\ P(o_1, o_1, l, o_1) \wedge P(o_2, o_2, l, o_2) \Rightarrow (o_1 = o_2)$

- Axiom 3: Names of attributes are unique in conjunction with the source object.
  $\forall\, o_1, x, l, y_1, o_2, y_2\ P(o_1, x, l, y_1) \wedge P(o_2, x, l, y_2) \Rightarrow (o_1 = o_2) \vee (l = in) \vee (l = isa)$

- Axiom 4: The name of instantiation and specialization objects (*in, isa*) is unique in conjunction with source and destination objects.
  $\forall\, o_1, x, l, y, o_2\ P(o_1, x, l, y) \wedge P(o_2, x, l, y) \wedge ((l = in) \vee (l = isa)) \Rightarrow (o_1 = o_2)$

- Axioms 5,6,7,8: Solutions for the predicates *In*, *Isa*, and A are derived from the object base.
  $\forall\, o, x, c\ P(o, x, in, c) \Rightarrow$ *In*(x, c)
  $\forall\, o, c, d\ P(o, c, isa, d) \Rightarrow$ *Isa*(c, d)
  $\forall\, o, x, n, y, p, c, m, d\ P(o, x, n, y) \wedge P(p, c, m, d) \wedge$ *In*$(o, p) \Rightarrow AL(x, m, n, y)$
  $\forall\, x, m, n, y\ AL(x, m, n, y) \Rightarrow A(x, m, y)$

- Axiom 9: An object x may not neglect an attribute definition in one of its classes.
  $\forall\, x, y, p, c, m, d$ *In*$(x, c) \wedge A(x, m, y) \wedge P(p, c, m, d) \Rightarrow$
  $\exists\, o, l\ P(o, x, l, y) \wedge$ *In*(o, p)

- Axioms 10,11,12: The *isa* relation is a partial order on the object identifiers.
  $\forall\, c$ *In*$(c,$ #Obj$) \Rightarrow$ *Isa*(c, c)
  $\forall\, c, d, e$ *Isa*$(c, d) \wedge$ *Isa*$(d, e) \Rightarrow$ *Isa*(c, e)
  $\forall\, c, d$ *Isa*$(c, d) \wedge$ *Isa*$(d, c) \Rightarrow (c = d)$

- Axiom 13: Class membership of objects is inherited upwardly to the superclasses.
  $\forall\, p, x, c, d$ *In*$(x, d) \wedge P(p, d, isa, c) \Rightarrow$ *In*(x, c)

- Axiom 14: Attributes are "typed" by their attribute classes.
  $\forall\, o, x, l, y, p\ P(o, x, l, y) \wedge$ *In*$(o, p) \Rightarrow \exists\, c, m, d\ P(p, c, m, d) \wedge$ *In*$(x, c) \wedge$ *In*(y, d)

- Axiom 15: Subclasses which define attributes with the same name as attributes of their superclasses must refine these attributes.
  $\forall\, c, d, a_1, a_2, m, e, f$
  $\mathit{Isa}(d, c) \wedge P(a_1, c, m, e) \wedge P(a_2, d, m, f) \Rightarrow \mathit{Isa}(f, e) \wedge \mathit{Isa}(a_2, a_1)$

- Axiom 16: If an attribute is a refinement (subclass) of another attribute then it must also refine the source and destination components.
  $\forall\, c, d, a_1, a_2, m_1, m_2, e, f$
  $Isa(a_2, a_1) \wedge P(a_1, c, m_1, e) \wedge P(a_2, d, m_2, f) \Rightarrow \mathit{Isa}(d, c) \wedge \mathit{Isa}(f, e)$

- Axiom 17: For any object there is always a unique "smallest" attribute class with a given label *m*.
  $\forall\, x, m, y, c, d, a_1, a_2, e, f\ (\mathit{In}(x, c) \wedge \mathit{In}(x, d) \wedge P(a_1, c, m, e) \wedge P(a_2, d, m, f)$
  $\Rightarrow \exists\, g, a_3, h\ \mathit{In}(x, g) \wedge P(a_3, g, m, h) \wedge \mathit{Isa}(g, c) \wedge \mathit{Isa}(g, d))$

- Axioms 18-22: Membership to the builtin classes is determined by the object's format.
  $\forall\, o, x, l, y\ (P(o, x, l, y) \Leftrightarrow \mathit{In}(o, \text{\#Obj}))$
  $\forall\, o, l\ (P(o, o, l, o) \wedge (l \neq in) \wedge (l \neq isa) \Leftrightarrow \mathit{In}(o, \text{\#Indiv}))$
  $\forall\, o, x, c\ (P(o, x, in, c) \wedge (o \neq x) \wedge (o \neq c) \Leftrightarrow \mathit{In}(o, \text{\#Inst}))$
  $\forall\, o, c, d\ (P(o, c, isa, d) \wedge (o \neq c) \wedge (o \neq d) \Leftrightarrow \mathit{In}(o, \text{\#Spec}))$
  $\forall\, o, x, l, y\ (P(o, x, l, y) \wedge (o \neq x) \wedge (o \neq y) \wedge (l \neq in) \wedge (l \neq isa) \Leftrightarrow \mathit{In}(o, \text{\#Attr}))$

- Axiom 23: Any object falls into one of the four builtin classes.
  $\forall\, o\ \mathit{In}(o, \text{\#Obj}) \Rightarrow \mathit{In}(o, \text{\#Indiv}) \vee \mathit{In}(o, \text{\#Inst}) \vee \mathit{In}(o, \text{\#Spec}) \vee \mathit{In}(o, \text{\#Attr})$

- Axioms 24-28: There are five builtin classes.
  $P(\#Obj, \#Obj, \text{Proposition}, \#Obj)$
  $P(\#Indiv, \#Indiv, \text{Individual}, \#Indiv)$
  $P(\#Attr, \#Obj, \text{attribute}, \#Obj)$
  $P(\#Inst, \#Obj, \text{InstanceOf}, \#Obj)$
  $P(\#Spec, \#Obj, \text{IsA}, \#Obj)$

- Axiom 29: Objects must be known before they are referenced. The operator $\preceq$ is a (predefined) total order on the set of identifiers.
  $\forall\, o, x, l, y\ P(o, x, l, y) \Rightarrow (x \preceq o) \wedge (y \preceq o)$

- Axioms 30,31 (axiom schemas): For any object $P(p, c, m, d)$ in the extensional object base we have two formulas for "rewriting" the *In* and A predicates. The *In* is mapped to a unary predicate where the class name is forming part of the predicate name and the A predicates is mapped to a binary predicate that carries the identifier of the class of the attribute in its predicate name. Internally, user-defined deductive rules that derive *In* and A predicates will also be rewritten accordingly. This extends the choices for static stratification.
  $\forall\, o\ \mathit{In}(o, p) \Rightarrow \mathit{In}.p(o)$
  $\forall\, o, x, l, y\ P(o, x, l, y) \wedge \mathit{In}(o, p) \Rightarrow A.p(x, y)$

ConceptBase allows to add user-defined rules and constraints. The semantics of an O-Telos database including such rules and constraints is the perfect model of the deductive database with the $P(o, x, l, y)$ as the only extensional predicate and all axioms and user-defined rules/constraints as deductive rules. Note that integrity constraints can be rewritten to deductive rules deriving the predicate *inconsistent*.

This list of axioms is excerpted from M.A. Jeusfeld: Änderungskontrolle in deduktiven Objektbanken. Dissertation Universität Passau, Germany, 1992. Available as Volume DISKI-19 from INFIX-Verlag, St. Augustin, Germany or via http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/d340216/diski19.pdf (in German).

Axioms 19-21 have been corrected after Christoph Radig found an example that led to the undesired instantiation of an individual object to #Inst or #Spec, respectively.

Appendix C

Graphical Types

The concept of a *graphical type* enables the specification of an external graphical presentation for ConceptBase objects. The graphical type is declared using a special pre-defined attribute category. An application program then uses this information to determine the graphical presentation of an object.

The next subsection introduces the basic concepts behind graphical types, while section C.2 presents the standard graphical type definitions for the ConceptBase Graph Editor. Section C.3 describes the definition of application-specific types.

C.1 The graphical type model

A specific graphical type is defined as an instance of the object `GraphicalType`. For the new Graph Editor implemented in Java, this class has been specialized in a class `JavaGraphicalType`. Instances of this class specify a graphical represenation of an object by defining graphical attributes such as shape, color, line thickness, font etc. Since the actual attributes and their admissible value depend on the used visualization tool, the definition of `GraphicalType` looks very simple.

```
GraphicalType in Class
end
```

The declaration of a graphical type for a concrete object is done by using the attribute `graphtype` which is defined for `Proposition` and therefore available for all objects:

```
Proposition with
  attribute
     graphtype : GraphicalType
end
```

The attribute can be defined explicitely for an object or can be specified by using a deductive rule (see section C.2 for an example). For the Graph Editor application included in the usage enviroment, one can attach a priority value to each graphical type. If there multiple graphtype attributes defined for one object, the graphical type with the highest priority value will be used.

Many applications employ multiple notations to provide different perspectives on the same set of objects. Each perspective emphasizes on a specific aspect of the world, such as the data oriented, the process oriented and the behavior oriented viewpoint, and uses an aspect-specific notation. A graphical notation (as e.g. the Entity-Relationship diagram) typically consists of a set of different graphical symbols (as e.g. diamonds, rectangles, and lines). A *graphical palette* is used to combine the set of graphical types that together form a notation:

```
Individual GraphicalPalette in Class with
  attribute
     contains : GraphicalType
     default : GraphicalType
end
```

In such a setting the same object may participate in different perspectives. ConceptBase offers the possibility to specify multiple graphical types for the same object. A tool can then provide different graphical views on the same object. To get the desired graphical type of an object under a specific palette, an application program specifies the name of the actual graphical palette as answer format when querying the ConceptBase server. Although this mechanism is available for arbitrary application programs we restrict our description to the ConceptBase Graph Editor.

The `default` specification serves as a catch all: an answer object, for which none of the graphical types of the current palette is specified, is presented using the default graphical type of that palette.

C.2 The standard graphical types

The Graph Editor is implemented using the Java Programming Language. It is entirely based on the Swing toolkit (package javax.swing). The graphical objects shown in the Graph Editor are all instances of the class JComponent in the javax.swing package. User-defined representations of objects can be provided by overwriting a specific class of the Graph Editor (details are given below).

C.2.1 The extended graphical type model

Based on our experience with the old graph browser for X11, we have extended the graphical type model for the Graph Editor. First, the class `GraphicalType` has been specialized by a class `JavaGraphicalType`:

```
Class JavaGraphicalType isA GraphicalType with
 attribute
   implementedBy : String;
   property : String;
   priority : Integer
 attribute,rule
   rPriority : $ forall jgt/JavaGraphicalType (not (exists i/Integer
                A_e(jgt,priority,i))) ==> A(jgt,priority,0) $
end

Individual DefaultIndividualGT in JavaGraphicalType with
  attribute,property
     bgcolor : "210,210,210";
     textcolor : "0,0,0";
     linecolor : "0,0,0";
     shape : "i5.cb.graph.shapes.Rect"
  attribute,implementedBy
     implBy : "i5.cb.graph.cbeditor.CBIndividual"
end
```

The object `DefaultIndividualGT` is an example for the instantiation of a graphical type. The attribute `implementedBy` specifies the full name of the Java class that provides the implementation for this graphical type. This class has to be a sub class of `"i5.cb.graph.cbeditor.CBUserObject"`. The `property` attribute specifies name-value pairs which will be used by the Java implementation to set certain properties, e.g. color, shape, font[1]. The priority value is used to resolve ambiguity if multiple graphical types apply to one object. The graphical type with the highest priority will be used. The rule specifies a default value of 0 for the priority.

The graphical palette has also been extended. There are now defaults for different types of objects, and graphical types for implicit links can be defined. Thus, the `default` attribute defined in `GraphicalPalette` will not be used anymore. The `contains` attribute has still to be used, i.e. a graphical type will only be used if it is also contained in the current graphical palette. Although the attributes are not declared

[1] Colors are given as RGB color value, e.g. 210,210,210 is light grey and 0,0,0 is black.

as `single` and `necessary`, each graphical palette should have at least one and at most one attribute in these categories.

```
Class JavaGraphicalPalette isA GraphicalPalette with
  attribute
     defaultIndividual : JavaGraphicalType;
     defaultLink : JavaGraphicalType;
     implicitIsA : JavaGraphicalType;
     implicitInstanceOf : JavaGraphicalType;
     implicitAttribute : JavaGraphicalType
end
```

C.2.2 Default graphical types

For the standard objects, there are a number of predefined graphical types. There are contained in the graphical palette `DefaultJavaPalette` which is used by default by the Graph Editor.

type of object	graphical type	style
Individuals	DefaultIndividualGT	gray box
Links	DefaultLinkGT	thin black line with label
InstanceOf	DefaultInstanceOfGT	green line without label
IsA	DefaultIsAGT	blue line without label
Attribute	DefaultAttributeGT	black line with label
Class	ClassGT	turquoise box
SimpleClass	SimpleClassGT	pink oval
MetaClass	MetaClassGT	light blue oval
MetametaClass	MetametaGT	bright green oval
QueryClass	QueryClassGT	red oval
Implicit In	ImplicitInstanceOfGT	dashed green line
Implicit IsA	ImplicitIsAGT	dashed blue line
Implicit Attribute	ImplicitAttributeGT	dashed black line

The object `DefaultJavaPalette` has also some rules which define the default relationship between objects and graphical types, e.g. all instances of `Class` have the graphical type `ClassGT`.

C.3 Customizing the graphical types

To support the user in defining his own graphical types we provide some examples and documentation of the properties.

There are two ways to customize the graphical types:

- Defining new graphical types with different properties using the provided implementations i5.cb.graph.cbeditor.CBIndividual and i5.cb.graph.cbeditor.CBLink
- Defining new graphical types with a different implementation class which extends i5.cb.graph.cbeditor.CBUserObject (or CBIndividual or CBLink)

Both possibilities will presented in the next two subsections.

C.3.1 Properties of CBIndividual and CBLink

The easiest way to modify the representation of an object in the Graph Editor is to load an existing graphical type, modify its properties and store it as a new graphical type.

The properties available and there meaning are given in the following. Note that colors have to be given as RGB color value, e.g. ”0,0,0” is black, ”255,0,0” is red, ”255,255,255” is white, etc. Furthermore, all attributes have to be strings, even if they are just numbers, e.g. use ”1” instead of 1 as attribute value.

bgcolor: Background color of the shape (default: transparent)

textcolor: Foreground color of the shape (i.e. text color) (default: black ”0,0,0”)

linecolor: Color of the border of the shape (default: transparent)

linewidth: Width of the border of the shape (default: ”1”)

edgecolor: Color of the edge (default: black ”0,0,0”)

edgewidth: Width of the edge (default: ”1”)

edgestyle: possible values are: ”continuous”, ”dashed”, ”dotted”, ”dashdotted” (default: ”continuous”)

shape: The name of the class representing the shape and implementing the interface `i5.cb.graph.shape.IGraphShape` (default: no shape). The package `i5.cb.graph.shape` defines some useful default shapes, see below for details. The shape will be drawn in the background of the small component. In the default implementation, the small component is a transparent JLabel, thus the shape is completely visible. Note, that this might not be the case if you are going to change the implementation of a graphical type (see below).

label: The label to be used for this object instead of the object name

align: Alignment of the label (possible values are ”center”, ”left”, and ”right”; default is ”center”)

size: Size of the node in pixels, e.g. ”20x20” (default: done automatically by Java LayoutManager)

font: Name of the font to be used for the shape (e.g., ”Arial”, default: Default font of Java)

fontsize: Size of the font in pixels (default: default font size of Java)

fontstyle: The style of the font (e.g., ”bold”, ”italic”, ”underlined”, ”bold,italic”, ...)

Do not forget to include the new graphical type into the graphical palette. It is not necessary to define a new graphical palette, you can extend the default palette. Furthermore, you have to define the `graphtype` attribute of some object in such a way that it refers to the new graphical type. Make sure, that the new graphical type has a higher priority than other graphical type which might apply (10 is the highest priority of the default graphical types).

An example of user-defined graphical types can be found in D.2.

C.3.2 Providing a new implementation for a graphical type

The `implementedBy` attribute of a graphical type specifies the class name of a Java class implementing this graphical type. This class has to be a subclass of “i5.cb.graph.cbeditor.CBUserObject”. If you are going to implement your own class, it is most useful to extend CBIndividual or CBLink in the package i5.cb.graph.cbeditor.

The class CBUserObject provides several methods which can be overwritten to implement your own graphical types:

Component getSmallComponent() returns the small component (i.e. the component which is shown first). The return value should be an instance of `javax.swing.JComponent` although only `java.awt.Component` is required as return value. AWT Component will probably not work correctly in the Swing-based Graph Editor. If the method returns null, then the result of getComponent() will be used to visualize the object.

Component getComponent() returns the main component for this user object. This component is used when the small component is not shown. To be compatible with the graph editor which is implemented in JFC/Swing, the component should be a subclass of JComponent. This method may return null, but then getSmallComponent must return a value.

Shape getShape() returns the shape for this component. Shape is defined in the package java.awt and represents any type of shape, e.g. polygon, rectangle, ellipse, etc. As said above, the shape is shown in the background of the small component, so it may not be visible if the small component is not transparent. The default implementation in CBIndividual and CBLink provide a transparent JLabel as small component so that the shape is visible. The Graph Editor contains several predefined shapes (see below).

boolean doCommit() This method is called when the user has clicked on the "Commit" button. Changes that have been made within this component (e.g. within a form) can then be added to the list of objects to be removed/added from the database. If the method returns false, then the commit operation will be aborted.

Furthermore, the CBUserObject class provides several methods which might be useful for implementing new graphical types:

String toString() returns the full object name of the Telos object which is represented by this user object

TelosObject getTelosObject() returns the Telos object which is represented by this user object (see documentation of package i5.cb.telos.object for more details on TelosObject)

CBFrame getCBFrame() returns the frame in which this object is presented (a CBFrame is an extension of a JInternalFrame)

ObjectBaseInterface getObi() returns the ObjectBaseInterface (i.e. a connection to the server) of the current frame. This is useful if you want to execute your own queries to retrieve additional information from the server. (see documentation of package i5.cb.telos.object and i5.cb.api for more details)

String getProperty(String property) returns the value of a property

boolean hasProperty(String property) returns true if the property is defined for this graphical type

JPopupMenu getPopupMenu() returns the popup menu for this object. By using this method you can extend the popup menu with own operations. If you overwrite this method, the operations in the default popup menu will not be available.

Example: The following example defines a graphical type that uses a JButton to visualize the object. Only the method getSmallComponent is overwritten. The background color of the button will be changed if the property `bgcolor` has been defined. The example shows that the implementation of graphical types is quite simple.

```
import i5.cb.graph.cbeditor.*;
import javax.swing.*;

import java.awt.Component;

/**
 * An example for a user defined CBUserObject
 */

public class MyOwnGraphType extends CBUserObject {

    public Component getSmallComponent() {
       JButton jButton=new JButton(this.getTelosObject().toString());
       if(hasProperty("bgcolor"))
          jButton.setBackground(CBUtil.stringToColor(getProperty("bgcolor")));
       return jButton;
    }
}
```

C.3.3 Shapes

The package i5.cb.graph.shape contains several shapes which might be useful for the ConceptBase Graph Editor. To use these shapes, you have to specify the full class name (including the package name) as value of the property `shape` of a graphical type.

class name	**style**
Arrow	Head of an arrow pointing to the right
Arrow2	Complete arrow pointing up
Cross	a cross (like the red cross)
Diamond	a diamond/rhombus
DoubleArrow	a horizontal arrow pointing in both directions
Ellipse	Ellipse
Hexagon, Octagon, Pentagon	as the name says
Rect	a rectangle
RoundRectangle	a rectangle with round corners
Star	a star
Triangle	a triangle
XCross	a cross in the form of an X

The class `GraphShapePolygon` is the base class for most of the shapes defined here and might be used to define your own shapes. Its constructor is identical with the constructor of java.awt.Polygon:

```
public GraphShapePolygon(int[] xs, int[] ys, int num)
```

`xs` and `ys` describe the coordinates of the corners of the polygon and `num` is the length of the array xs or ys. Start and end point should be identical. For example, to define a triangle, you need the following code:

```
import i5.cb.graph.shape.GraphShapePolygon;

public class Triangle extends GraphShapePolygon {
    private static int xpoints[]={0,2,4,0};
    private static int ypoints[]={2,0,2,2};

    public Triangle() {
        super(xpoints,ypoints,4);
    }
}
```

Appendix D

Examples

D.1 Example model: The Employee model

The Employee model can be found in the directory `$CB_HOME/examples/QUERIES/`. It consists out of the following files:

Employee_Classes.sml: The class definition

Employee_Instances.sml: Some instances for this model

Employee_Queries.sml: Queries for this model

Note, that the files must be loaded in this order into the server.

D.2 A Telos modeling example - ER diagrams

D.2.1 The basic model

This example gives a first introduction into some features introduced in ConceptBase version 4.0. It demonstrates the use of *meta formulas* and *graphical types* while building a Telos model describing Entity-Relationship-Diagrams. The following model forms the basis:

```
{************************}
{*                      *}
{* File: ERModelClasses *}
{*                      *}
{************************}

Class Domain
end

Class EntityType with attribute
     eAttr : Domain;
     keyeAttr : Domain

end

Class RelationshipType with attribute
     role : EntityType
end
```

```
Class MinMax
end

"(1,*)" in MinMax with
end

"(1,1)" in MinMax with
end

Attribute RelationshipType!role with attribute
     minmax: MinMax
end
```

The model defines the concepts of *EntityTypes* and *RelationshipTypes*. Each entity that participates in a relationship plays a particular *role*. This role is modelled as a Telos *attribute-link* of the object *RelationshipType*. The *attributes* describing the entities are modelled as Telos *attribute-links* to a class *Domain* containing the value-sets. Roles can be restricted by the "(min,max)"-constraints "*(1,*)*" or "*(1,1)*". The next model contains a concrete ER-model.

```
{************************}
{*                      *}
{* File: Emp_ERModel    *}
{*                      *}
{************************}

Class Employee in EntityType with
  keyeAttr,attribute
     ssn : Integer
  eAttr,attribute
     name : String
end

Class Project in EntityType with
  keyeAttr,attribute
     pno : Integer
  eAttr,attribute
     budget : Real
end

Integer in Domain end

Real in Domain end

String in Domain end

Class WorksIn in RelationshipType  with role,attribute
     emp : Employee;
     prj : Project
end

WorksIn!emp with minmax
     mProjForEmp: "(1,*)"
end

WorksIn!prj with minmax
     mEmpForProj: "(1,*)"
end
```

```
ConceptBase in Project with pno
     cb_pno : 4711
end

Martin in Employee with ssn
      martinSSN : 4712
end

M_CB in WorksIn with
   emp
      mIsEmp : Martin
   prj
      cbIsPrj : ConceptBase
end

Hans in Employee with ssn
     hans_ssn : 4714
end
```

The entity-types *Employee* and *Project* participate in a binary relationship *WorksIn*. The attributes `Employee!ssn` and `Project!pno` are key-attributes of the respective objects.

D.2.2 The use of meta formulas

The above model distinguishes *attributes* and *key attributes*. One important constraint on key attributes is monovalence. In the previous releases of ConceptBase it was possible to declare Telos-attributes as instance of the attribute-categories *single* or *necessary*, but the constraint ensuring this property could not be formulated in a general manner, because the use of variables as placeholders for Telos-classes e.g in an *In*-Literal was prohibited. To overcome this restriction, *meta formulas* have been integrated into the system. An assertion is a *meta formula* if it contains such a class-variable. The system tries to replace this *meta formula* by a set of semantic equivalent formulas which contain no class-variables. In previous releases properties as *single* or *necessary* had to be ensured "manually" by adding a constraint for each such attribute. This job is now performed automatically by the system.

Example: *necessary* and *single*

The following meta formula ensures the *necessary* property of attributes, which are instances of the category `Proposition!necessary`. The semantics of this property is, that for every instance of the source class of this attribute there must exist an instantiation of this attribute.

```
Class with constraint, attribute
        necConstraint:
       $ forall c,d/Proposition p/Proposition!necessary x,m/VAR
            P(p,c,m,d) and (x in c) ==>
             exists y/VAR (y in d) and (x m y) $
end
```

It reads as follows:

> For each attribute with label `m` between the classes `c` and `d`, which instantiates the attribute `Proposition!necessary` and for each instance `x` of `c` there should exist an instance `y` of `d` which is destination of an attribute of `x` with category `m`.

One should notice that the predicates `In(x,c)` and `In(y,d)` cause this formula to be a meta formula. The instantiation of `x` and `m` to the class VAR is just a syntactical construct. Every variable in a constraint

has to be bound to a class. This restriction is somehow contrary to the concept of meta formulas and the VAR-construct is a kind of compromise. The resulting `In`-predicates are discarded during the processing of meta formulas. The VAR-construct is only allowed in meta formulas. It enables the user to leave the concrete classes of `x` and `m` open, without instantiating them to for example to *Proposition*.

The *single*-constraint can be defined in analogy. These constraints can be added to the system as if they were "normal" constraints. Their effect becomes visible, when declaring attributes as *necessary* or *single*. This is done in the following model.

```
{************************}
{*                      *}
{ File: ERSingNec       *}
{*                      *}
{************************}

{* necessary constraint (metaformula) *}
Class with constraint
    necConstraint:
    $ forall c,d/Proposition p/Proposition!necessary x,m/VAR
            P(p,c,m,d) and (x in c) ==>
             exists y/VAR (y in d) and (x m y) $
end




{* every Entity has a key *}
Class EntityType with
  necessary
     keyeAttr : Domain
end


{* single constraint (metaformula) *}
Class with constraint
    singleConstraint :
    $ forall c,d/Proposition p/Proposition!single x,m/VAR
              P(p,c,m,d) and (x in c) ==>
                (
                  forall a1,a2/VAR
                    (a1 in p) and (a2 in p) and Ai(x,m,a1) and Ai(x,m,a2) ==>
                   (a1=a2)
                ) $
end

{* every Entity key is monovalued ( = necessary and single)*}
Class EntityType with rule
     keys_are_necessary:
        $forall a/EntityType!keyeAttr In(a,Proposition!necessary)$;
     keys_are_single:
        $forall a/EntityType!keyeAttr In(a,Proposition!single)$
end
```

The effects of this transaction can be shown by displaying the instances of instances of the class *metaMSFOLconstraint*. The *single*- and *necessary* constraints are inserted into this class after adding them to the system. These constraints themselves can have pecializations: constraints which are added automatically to the system when inserting objects into the attribute-category *single* resp. *necessary*.

For the ER-example, one of the created formulas reads as:

```
$ forall x/EntityType (exists y/Domain   (x keyeAttr y)) $
```

bserve the relationship to the *necessary*-formula: The formula has been generated by computing one extension of `In(p,Proposition!necessary) and P(p,c,m,d)` and replacing the predicates `In(p, Proposition!necessary` and `P(p,c,m,d)` by this extension, which results in the following substitution for the remaining formula:

c	EntityType
d	Domain
m	keyeAttr

Metaformulas defining sets of rules

Another use of *Metaformulas* is the formulation of deductive rules. Metaformulas defining deductive rules extend the possibilities of defining derived knowledge.

Assignment of graphical types: This example first demonstrates the use of meta formulas to assign graphical types to object-categories. The minimal graphical convention for ER-diagrams is to use rectangular boxes for entities and diamond-shaped boxes for relationships. In our modelling example these graphical types are assigned to objects which are instances of *EntityType* or *RelationshipType* and to instances of these objects.

```
{*****************************************}
{*                                       *}
{* File: ERModelGTs                      *}
{* Definition of the graphical palette for *}
{* ER-Diagrams for use on color displays   *}
{*                                       *}
{*****************************************}


{* graphical type for inconsistent roles *}
Class InconsistentGtype in JavaGraphicalType with
  attribute,property
     textcolor : "0,0,0";
     edgecolor : "255,0,0";
     edgewidth : "2"
implementedBy
     implBy : "i5.cb.graph.cbeditor.CBLink"
priority
     p : 14
end


{* graphical type for entities *}
Class EntityTypeGtype in JavaGraphicalType with
property
     bgcolor : "10,0,250";
     textcolor : "0,0,0";
     linecolor : "0,55,144";
     shape : "i5.cb.graph.shapes.Rect"
implementedBy
     implBy : "i5.cb.graph.cbeditor.CBIndividual"
priority
    p : 12
end

{* graphical type for relationships *}
Class RelationshipGtype in JavaGraphicalType with
```

```
property
     bgcolor : "255,0,0";
     textcolor : "0,0,0";
     linecolor : "0,0,255";
     shape : "i5.cb.graph.shapes.Diamond"
implementedBy
     implBy : "i5.cb.graph.cbeditor.CBIndividual"
priority
    p : 13
end


{* graphical palette *}
Class ER_GraphBrowserPalette in JavaGraphicalPalette with
  contains,defaultIndividual
     c1 : DefaultIndividualGT
  contains,defaultLink
     c2 : DefaultLinkGT
  contains,implicitIsA
     c3 : ImplicitIsAGT
  contains,implicitInstanceOf
     c4 : ImplicitInstanceOfGT
  contains,implicitAttribute
     c5 : ImplicitAttributeGT
  contains
     c6 : DefaultIsAGT;
     c7 : DefaultInstanceOfGT;
     c8 : DefaultAttributeGT;
     c14 : EntityTypeGtype;
     c15 : RelationshipGtype;
     c16 : InconsistentGtype
end


EntityType with rule
    EntityGTRule:
        $ forall e/EntityType  A(e,graphtype,EntityTypeGtype)$ ;
    EntityGTMetaRule:
        $ forall x/VAR (exists e/EntityType In(x,e)) ==>
          A(x,graphtype,EntityTypeGtype)$
end


RelationshipType with rule
     RelationshipGTRule:
         $ forall r/RelationshipType A(r,graphtype,RelationshipGtype)$ ;
     RelationshipGTMetaRule:
         $ forall x/VAR (exists r/RelationshipType In(x,r))  ==>
           A(x,graphtype,RelationshipGtype) $
end
```

To activate the *ER_GraphBrowserPalette* , select this graphical palette when you start the Graph Editor or make a new connection in the Graph Editor (see section 8.2).

Handling inconsistencies The *necessary* and *single* conditions on attributes from the previous section could also be expressed as deductive rule. The difference is, that if they are formulated as constraints, every transaction violating the constraint would be rejected. The definition of rules handling *necessary* and *single* enables the user to handle inconsistencies in his model. The following example demonstrates this

concept in the context of our ER-model. The example defines rules handling the restriction of roles by the "(min,max)"-constraint "*(1,*)*". This restriction is not implemented using constraints. Instead a new Class *Inconsistent* is defined, containing all role-links which violate the "*(1,*)*" constraint. These inconsistent links get a different graphical type (e.g a red coloured attribute link) than consistent role links to visualize the inconsistency graphically.

```
{****************************************}
{*                                      *}
{* File: ERIncons                       *}
{* Definition of a Class "Inconsistent" *}
{* containing roles violating the "(1,*)"  *}
{* constraint                           *}
{*                                      *}
{****************************************}

{* new attribute category revNec for attributes
   which are "reverse necessary" *}
Class with attribute
           revNec : Proposition
end

{* roles with "(1,*)" must fullfill revNec property *}
RelationshipType with rule, attribute
    revNecRule:
    $ forall ro/RelationshipType!role
        A(ro,minmax,"(1,*)")  ==> In(ro,Class!revNec) $
end

{* definition of Class "Inconsistent" *}
{* forall instances "p" of Class!revNec:
   If there exists a destination class "d", there must
   be a source class "c" with an attribute instantiating "p",
   otherwise "p" is inconsistent
*}
Class Inconsistent with rule, attribute
   revNecInc:
   $ forall p/Class!revNec
       (exists c,m,d/VAR  y/VAR  P(p,c,m,d) and In(y,d) and
          not(exists x/VAR  In(x,c) and A(x,m,y))) ==>
          In(p,Inconsistent)$
end
```

To activate the different graphical representation of inconsistent roles, the definition of the graphical type for attributes has to be modified. Tell the following frame:

```
Inconsistent with rule,attribute
     incRule :
        $ forall e/Attribute In(e,Inconsistent) ==>
          (e graphtype InconsistentGtype)$
end
```

The effect of these transactions can be shown when starting the *Graph Editor* and displaying the attributes of the `RelationshipType` instance `WorksIn`. Be sure to switch the graph editor to the palette `ER_GraphBrowserPalette` before doing so (see section 8.2). The attribute `WorksIn!emp` is displayed as red link like in figure D.1. If you had already started the graph editor before telling the last frame, you should synchronize it with the CBserver via the menu Current connection. By telling

```
H_CB in WorksIn with
```

```
    emp
       hIsEmp : Hans
    prj
       cbIsPrj : ConceptBase
end
```

the inconsistency is removed from the model. To see the update of the graphical type in the Graph Editor, you have to select the inconsistent link and select "Validate and update selected objects" from the "Current connection" menu.

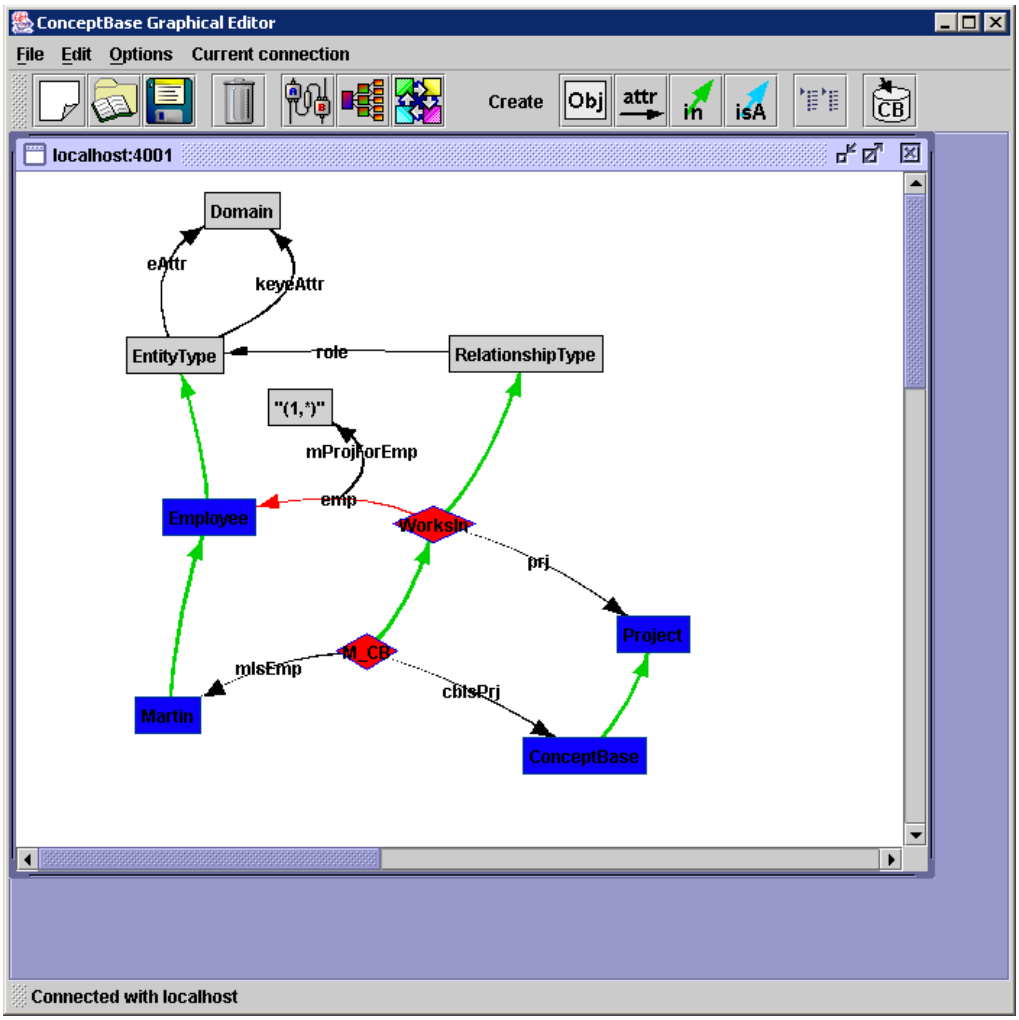

Figure D.1: Graph Editor showing the example model with inconsistent link

D.2.3 Limitations and final remarks

Metaformulas extend the expressive power of defining rules and constraints for ConceptBase Objectbases. The implementation of this mechanism is not complete at the moment, but should enable the use of the most frequently requested concepts like *single* and *necessary*. Some limitations are listed below:

- **limited partial evaluation**
  The partial evaluation procedure is limited to a conjunction of predicates, preceeded by *forall*-quantors.

- **no source-to-source transformation**
  Formulas are not converted automatically into the form mentioned above supported by the meta formula mechanism, even if they could be transformed. If the class-variables can't be bound using partial evaluation of the input-formula, the formula is rejected, even if there exists an equivalent formula, in which partial evaluation could be used to bind the class-variables.

- **not all classes are supported**
  Generated formulas where variables are quantified over instances of the following classes or attributes of them will be ignored. Those classes are:
  Boolean,Integer,Real,String,TransactionTime,
  MSFOLassertion,MSFOLrule,MSFOLconstraint,
  metaMSFOLconstraint,metaMSFOLrule,
  BDMConstraintCheck,BDMRuleCheck,LTruleEvaluator,ExternalReference,
  QueryClass, BuiltinQueryClass,AnswerRepresentation,
  GraphicalType,X11_Color,ATK_TextAlign,
  ATK_Fonts,ATK_LineCap,ATK_ShapeStyle
  The justification is twofold. Some of these generated formulas, e.g. a formula beginning with

  ```
  $ forall x/1 ...$
  ```

  are regarded as redundant, because the object *1* as instance of *Integer* should have no instances. Another justification is, that the use of meta formulas should be restricted to user-defined modeling tasks. Manipulation of the most system classes is disabled for reasons of efficiency and safety.

In the case of deductive rules, additional problems arise similar to the straticfication problem. At the moment only monotonous transactions are allowed. This means that generated formulas can only be inserted or deleted during one transaction, both operations at the same time are not permitted.

The meta formula mechanism also influences the efficiency of the system: every transaction has to be supervised wether it affects the meta formulas or one of the generated formulas. If the preconditions of the generated formulas don't hold anymore, e.g. if the instances of `EntityType!eAttr` are no longer instances of `Proposition!necessary` in the previous model, the corresponding generated formula has to be deleted. If additional attributes are instantiated to `Proposition!necessary`, additional formulas have to be created. The process of supervising the transactions is quite expensive and if it slows down the overall performance too much, some of the meta formuals can be disabled temporarily (Untelling a meta formula removes all the code generated).

Many more examples for meta formulas, e.g. for defining transitivity of attributes, are available from the CB-Forum (http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/1042523).

Appendix E

Predefined Query Classes

This chapter gives an overview of the query classes which are predefined in a standard ConceptBase installation. The names of parameters of the queries are set in `typewriter` font. Most of the queries listed here are used by the ConceptBase user interface CBiva to interaction with the CBserver. A normal user typically formulates queries herself. In fact, most queries listed below are very simple and directly representation as query class. An exception are the functions for computation and counting. They cannot be expressed by simple query classes but extend the expressiveness of the system.

E.1 Query classes and generic query classes

These queries can also be used in the constraints of other queries.

E.1.1 Instances and classes

ISINSTANCE: Checks whether `obj` is instance of `class`. The result is either `TRUE` or `FALSE`.

IS_EXPLICIT_INSTANCE: Same as before, but returns `TRUE` only if `obj` is an *explicit* instance of `class`.

find_classes: Lists all objects of which `objname` is an instance.

find_instances: Lists all instances (implicit and explicit) of `class`.

find_explicit_instances: Same as before, but only explicit instances are returned.

E.1.2 Specializations and generalizations

ISSUBCLASS: Checks whether `sub` is subclass of `super`. The result is either `TRUE` or `FALSE`.

IS_EXPLICIT_SUBCLASS: Same as before, but returns `TRUE` only if `sub` is an *explicit* subclass of `super`.

find_specializations: Lists the subclasses of `class`. If `ded` is `TRUE`, then the result will also include implicit subclasses, if `ded` is `FALSE` only explicit information will be included.

find_specializations: Same as before, but for super classes.

E.1.3 Attributes

IS_ATTRIBUTE_OF: Returns `TRUE` if `src` has an attribute of the category `attrCat` which has the value `dst`.

IS_EXPLICIT_ATTRIBUTE_OF: Same as before, but only for explicit attributes.

find_all_explicit_attribute_values: Lists all attribute values of `objname` that are explicitely defined.

find_iattributes: Lists the attributes that *go into* `class`.

find_referring_objects: Lists the objects that have an explicit attribute link to `class`.

find_referring_objects2: Lists the objects that have an explicit attribute link to `objname` and for which the attribute link is an instance of `cat`.

find_all_referring_objects2: Same as before, but including implicit attributes.

find_attribute_categories: Lists all the attribute categories that may be used for `objname`. This is a lookup of all attributes of all classes of `objname`.

find_incoming_attribute_categories: In contrast to the previous query, this query returns all attribute categories that go into `objname` (i.e. attribute categories for which `objname` can be used as an attribute value).

find_attribute_values: Lists all objects that are attribute values of `objname` in the attribute category `cat`.

find_explicit_attribute_values: Same as before, but only for explicit attributes.

E.1.4 Links between objects

find_incoming_links: Lists the links that *go into* `objname` and are instance of `category`. Note that all types of links are returned, including attributes, instance-of-links and specialization links.

find_incoming_links_simple: Same as before, but without the parameter `category`.

find_outgoing_links: Lists the links that *come out of* `objname` and are instance of `category`. Note that all types of links are returned, including attributes, instance-of-links and specialization links.

find_outgoing_links_simple: Same as before, but without the parameter `category`.

get_links2: Return the links between `src` and `dst`.

get_links3: Return the links between `src` and `dst` that are instance of `cat`.

E.1.5 Other queries

find_object: This query just returns the object given as parameter `objname`, if it exists. Thus it can be used to check whether `objname` exists, but there is a builtin query `exists` which does the same. The query is mainly useful in combination with a user-defined answer format (e.g. the Graph Editor is using this query to retrieve the graphical representation of the object).

AvailableVersions: Lists the instances of `Version` with the time since when they are know. This query is used by the user interface to use a different rollback time (Options → Select Version).

listModule: Lists the the content of a module as Telos frames, see also section 5.7.

E.2 Functions

Functions may also be used within other queries. You may define your own functions (see section 2.5).

E.2.1 Computation and counting

COUNT: counts the instances of a class, this may be also a query class

SUM: computes the sum of the instances of a class (must be reals or integers)

AVG: computes the average of the instances of a class (must be reals or integers)

MAX: gives the maximum of the instances of a class (wrt. the order of $<$ and $>$, see section 2.2)

MIN: gives the minimum of the instances of a class (wrt. the order of $<$ and $>$, see section 2.2)

COUNT_Attribute: counts the attributes in the specified category of an object

SUM_Attribute: computes the sum of the attributes in the specified category of an object (must be reals or integers)

AVG_Attribute: computes the average of the attributes in the specified category of an object (must be reals or integers)

MAX_Attribute: gives the maximum of the attributes in the specified category of an object (wrt. the order of $<$ and $>$, see section 2.2)

MIN_Attribute: gives the minimum of the attributes in the specified category of an object (wrt. the order of $<$ and $>$, see section 2.2)

PLUS: computes the sum of two reals or integers

MINUS: computes the difference of two reals or integers

MULT: computes the product of two reals or integers

DIV: computes the quotient of two reals or integers

IPLUS: computes the sum of two integers; result is an integer number

IMINUS: computes the difference of two integers; result is an integer number

IMULT: computes the product of two integers; result is an integer number

IDIV: computes the quotient of two integers and then truncates the quotient to the largest integer number smaller than or equal to the quotient

ConceptBase realizes the arithmetic functions via its host Prolog system SWI-Prolog. Integer numbers on the platforms Solaris-Intel, Solaris-Sparc and Linux are represented as 64-bit numbers, yielding a maximum range from $-2^{64}-1$ to $2^{64}-1$. SWI-Prolog supports by default under Windows and Linux64 arbitrarily long integers. Real numbers are implemented by SWI-Prolog as 64-bit double precision floating point numbers. ConceptBase uses 12 decimal digits.

E.2.2 String manipulation

ConcatenateStrings(3/4): concatenates two (or more) string objects

StringToLabel: removes the quotes of a string and returns it as a label (not an object), useful if labels should be passed as a parameter of a query

E.3 Builtin query classes

These queries must not be used within other queries as they do not return a list of objects. They may only be used directly from client programs.

exists: Checks whether `objname` exists and returns `yes` or `no`.

get_object: Returns the frame representation of `objname`. This query may be either called with just one parameter (`objname`) or with four parameters (`objname`, `dedIn`, `dedIsa`, `dedWith`). The `ded*`-parameters are boolean flags that indicate whether implicit (deduced) information should also be included in the frame representation. Note that the order of the parameters hast to be the same as listed above.

get_object_star: Returns the frame representation for all objects with a label that match the given wildcard expression. Only simple wildcards with a star (*) at the end are allowed.

rename: Renames an object from `oldname` to `newname`. This is a low-level operation directly on the symbol table that works directly on the symbol table. It only checks whether *newname* is not already used as label for a different object, no other consistency checks are performed. The parameters have to be given in the order `newname`, `oldname`.

Appendix F

CBserver Plug-Ins

Users with Prolog experience may find it interesting to use the LPI ("logic plug-in") feature of ConceptBase server. An LPI plug-in is essentially a small Prolog program that is inserted into the ConceptBase server code at run-time. It extends the functionality of the CBserver, for example for user-defined builtin queries.

The implementation of pre-defined builtin queries (see appendix E) is part of the CBserver.

You can create a file like `MyPlugin.swi.lpi` to provide the implementation for your self-defined builtin queries or for action calls in active rules (see section 4). You can use the full range of functions provided by the underlying Prolog system (here: SWI-Prolog, `http://www.swi-prolog.org`) and the functions of the CBserver to realize your implementation. You may consult the the CB-Forum for some examples at `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2768063`. You find there examples for sending emails from active rules, and for extending the set of builtin queries and functions.

F.1 Defining the plug-in

Once you have coded your file MyPlugin.swi.lpi, there are two ways to plug it into the CBserver. The first way is to copy the file into an existing database directory created by the CBserver.

```
CBserver -d MYDB
[load Telos definitions; then stop the CBserver]
cp MyPlugin.swi.lpi MYDB
```

This option makes the definitions only visible to a CBserver that loads the database MYDB. The second option is to instruct ConceptBase to load your LPI code to *any* database created by the CBserver. To do so, you just have to copy the LPI file into the directory with the system definitions:

```
cp MyPlugin.swi.lpi <CB_HOME>/lib/system
```

where `CB_HOME` is the directory into which you installed ConceptBase. The number of LPI files is not limited. You may code zero, one or any number of plug-in files.

A couple of useful LPI plug-ins are published via the CB-Forum, see `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2768063`. Note that these plugins are copyrighted typically come with their own license conditions that may be different to the license conditions of ConceptBase. If you plan to use the plugins for commercial purposes, you may have to acquire appropriate licenses from the plugin's authors.

F.2 Calling the plug-in

There are two ways to trigger the call of a procedure implemented by an LPI plugin.

1. By explicitely calling a builtin query class (or function) whose code has been implemented by the LPI plugin. The LPI code would then look similar to the code in SYSTEM.SWI.builtin and you must have defined an instance of `BuiltinQueryClass` that matches the signature of the LPI code. The call to the builtin query class may be enacted from the user interface, or it may be included as an `ASKquery` call in an instance of `AnswerFormat`. Refer for more information to section 3.2.5 and to the directory `Examples/BuiltinQueries` in your ConceptBase installation directory.

2. By calling the implemented function as a CALL action of an active rule. See section 4.2.2 for an example. In that case, there does not need to be a definition of a builtin query class (or function) to declare the signature of the procedure.

If the code of an LPI plugin realizes a function, e.g. selecting the first instance of a class, then you can use that function whereever functions are allowed. As an example, consider the definition of the LPI plugin `selectfirst.swi.lpi` from http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/d2984654/selectfirst.swi.lpi.txt:

```
compute_selectfirst(_res,_class,_c1) :-
      nonvar(_class),
      cbserver:ask([In(_res,_class)]),
      !.

tell: 'selectfirst in Function isA Proposition with
parameter class: Proposition end'.
```

The first clause is the Prolog code. The predicate must start with the prefix `compute_` followed by the name of the function. The first argument is for the result of the function. Subsequently, each input parameter is represented by two arguments, one for the input parameter itself and a second as a placeholder of the type of the input parameter. The second clause tells the new function as Telos object so that it can be used like any other Telos function. For technical reasons, the 'tell' clause may not span over more than 5 lines. Use long lines if the object to be defined is large.

If you just want to invoke the procedure defined in an LPI plugin via the CALL clause of an active rule, you do not need to include a 'tell' clause. Consider for example the SENDMAIL plugin from http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/2269675.

In all cases you need to be very careful with testing your code. Only use this feature for functions that cannot be realized by a regular query class or by active rules. LPI code has the full expressiveness of a programming language. Program errors may lead to crashes of the CBserver or to infinite loops or to harmful behavior such as deletion of files. Query classes, deductive rules and integrity constraints can never loop infinitely and can (to the best of our knowledge) only produce answer, not changes to your file system or interact with the operating system. Active rules could loop infinitely but also shall not change your file system and shall not interact with the operating system unless you call such code explicitely in the active rule.

You can disable the loading of LPI plugins with the (otherwisely undocumented) CBserver option `-g nolpi`. The CBserver will then not load LPI plugins upon startup. This option might be useful for debugging or for disabling loading LPI plugins that are configured in the `lib/system` sub-directory of your ConceptBase installation.

F.3 Programming interface for the plug-ins

The CBserver plug-ins need to interface to the functionalities of the CBserver, the Prolog runtime, and possibly the operating system. To simplify the programming of the CBserver plug-ins, we document here the interface of the module `cbserver`. We assume that the code for the plug-ins is written in SWI-Prolog and the user is familiar with the SWI-Prolog system.

cbserver:ask(Q,Params,A) asks the query `Q` with parameters `Params` to the CBserver. The answer is returned as a Prolog atom in `A`.
Example: `cbserver:ask(find_classes,[bill/objname],A)`

cbserver:ask(Preds) evaluates the predicates in list `Preds`. The predicate can backtrack and will bind in case of success the free variables in `Lits`. We currently support only the following predicates: `In`, `A`, `AL`, and `Isa`.
Example: `cbserver:ask([In(X,Employee),A(X,salary,1000)])`

cbserver:askAll(X,Lits,Set) finds all objects `X` that satisfy the condition in `Lits` and puts the result into the list `Set`. `A`, `AL`, and `Isa`.
Example: `cbserver:askAll(X,[In(X,Employee),A(X,salary,1000],S)`

cbserver:tellFrames(F) tells all frames contained in the atom `F`. The call will fail if there is any error in `F`.
Example: `cbserver:tellFrames('bill in Employee end')`

cbserver:makeName(Id,A) converts an object identifier to a readable object name.

cbserver:makeId(A,Id) converts an object name (Prolog atom) into an object identifier.

cbserver:arg2val(E,V) transforms an argument (either an object identifier or a functional expression) to a value (a number or a string). The value can then be used in Prolog style computations such as arithmetic expressions.

cbserver:val2arg(V,I) transforms a Prolog value (number, string) to an object identifier, possibly by creating a new object for the value.

cbserver:concat(X,Y) concats the strings (Prolog atoms) contained in the list `X`. The result is returned in `Y`.

Note that ConceptBase internally manages concepts by their object identifier. The programming interface instead addresses concepts (and objects) by their name, i.e. the label of the object or the Prolog value corresponding to the label. You may have to use the procedure `makeName` and `makeId` to switch between the two representations. The two procedures `arg2val` and `val2arg` are useful for defining new builtin functions on the basis of Prolog's arithmetic functions. Assume for example, that the object identifier `id123` has been created to correspond to the real number `1.5`. Then, the following relations hold: `makeName(id123,'1.5')`, `arg2val(id123,1.5)`. Hence, `makeName` returns the label `'1.5'` whereas `arg2val` returns the number 1.5.

The interface shall be extended in the future to provide more functionality. Be sure that you only use this feature if user-defined query classes cannot realize your requirements!